

AD-A110 196

MITRE CORP BEDFORD MA

F/6 9/2

A QUANTIFIABLE METHODOLOGY FOR SOFTWARE TESTING: USING PATH ANALYSIS

DEC 81 S PHOHA

F19628-81-C-0001

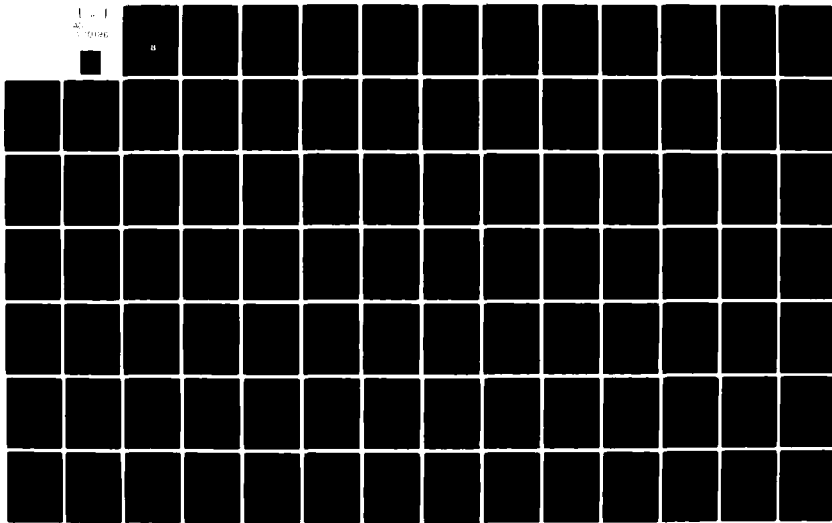
UNCLASSIFIED

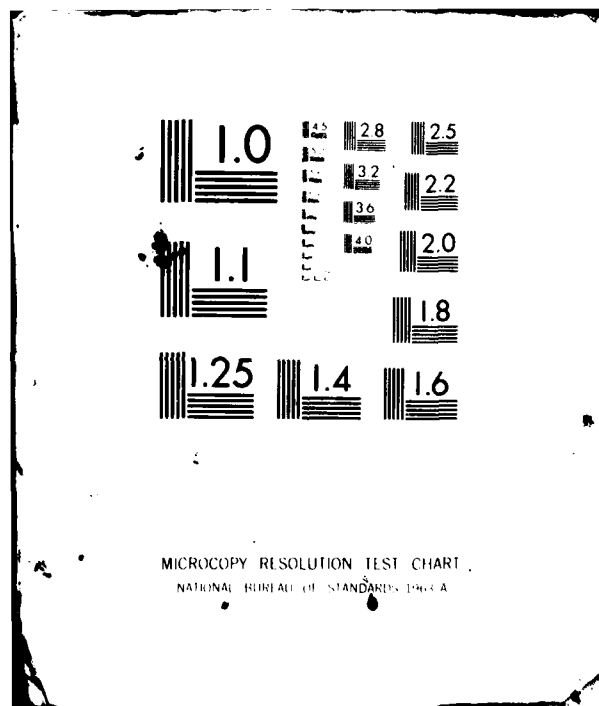
MTR-8393

ESD-TR-81-259

NL

1 - 1
10/19/86





ESD-TR-81-259

MTR-8393

**A QUANTIFIABLE METHODOLOGY FOR SOFTWARE TESTING:
USING PATH ANALYSIS**

By

SHASHI PHOHA

DECEMBER 1981

Prepared for

**DEPUTY FOR SURVEILLANCE AND CONTROL SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
Hanscom Air Force Base, Massachusetts**



Approved for public release; distribution unlimited.

Project No. 4130
Prepared by

**THE MITRE CORPORATION
Bedford, Massachusetts**

Contract No. AF19628-81-C-0001

82 8 006

231-050

AD A110196

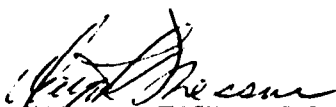
DTIC FILE COPY

When U.S. Government drawings, specifications, or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise, as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

Do not return this copy. Retain or destroy.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for public release; distribution unlimited.



WAYNE K. MESSNER, Colonel, USAF
Director
Physical Security Systems Directorate



HENRY D. SHAPIRO, GS-13
Chief, Entry Control Division
Physical Security Systems Directorate

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-81-259	2. GOVT ACCESSION NO. AD-A110 196	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A QUANTIFIABLE METHODOLOGY FOR SOFTWARE TESTING: USING PATH ANALYSIS		5. TYPE OF REPORT & PERIOD COVERED
7. AUTHOR(s) Shashi Phoha		6. PERFORMING ORG. REPORT NUMBER MTR-8393
9. PERFORMING ORGANIZATION NAME AND ADDRESS The MITRE Corporation P.O. Box 208 Bedford, MA 01730		8. CONTRACT OR GRANT NUMBER(s) AF19628-81-C-0001
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Surveillance and Control Systems Electronic Systems Division, AFSC Hanscom Air Force Base, MA 01731		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Project No. 4130
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE DECEMBER 1981
		13. NUMBER OF PAGES 92
		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) PATH ANALYSIS SOFTWARE TEST METRICS TEST EFFECTIVENESS MEASURES TEST SPECIFICATIONS		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This report is a comprehensive presentation of a quantitative methodology for software testing which measures test effectiveness at several different levels of program coverage and establishes confidence levels in the correctness of the program at these levels. Based on the resulting numerical specifications for testing a computer program, quantitative acceptance criteria are developed. These metrics are sensitive to cost and software criticality factors. The methodology, based on path analysis, is a natural extension of software engineering techniques to quality assurance for (over)		

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

20 (Concluded)

well-structured programs. It has been applied successfully, but several practical problems still remain. Application of this methodology to a software development program will provide control and visibility into the structure of the program and may result in improved reliability and documentation. Especially for the Air Force, when it acts only as a monitor, external to the software development process, the methodology provides a framework for proper planning and optimal allocation of test resources by quantifying the effectiveness of a test program and pre-determining the amount of testing required for achieving test objectives.

With the proof of the fundamental theorem of program testing in 1975, which establishes testing as the equivalent of a proof of correctness for programs which satisfy some structural constraints, systematic testing has become possibly the only effective means to assure quality of a program of non-trivial complexity. Thus, one may expect that the test methodology reported here, if applied to the problem of formal verification of computer aided software design specifications, may contribute significantly to a formal proof of correctness for computer security software in operating systems.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

ACKNOWLEDGMENTS

This report has been prepared by The MITRE Corporation under Project No. 4130. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts.

Thanks are extended to John H. James, Stuart M. Jolly, Charles M. Plummer, and William M. Stein for their constructive comments which helped improve this presentation. Thanks are also due to Lynne R. Wagstaff for her excellent secretarial support and Marina E. Pagoaga for her word processing support during the production of this report.

Accession For	
NTIS	GPA&I
DTIC	TAV
Unannounced	
Justification	
By	
Dist	
A	



Since modern experimental physical science started with the work of Bacon, no one who published quantitative results without a credible estimate of accuracy would have been taken seriously. In computing work, much of what we do amounts to numerical experiments...yet it has become commonplace for computer users who are otherwise competent scientists to generate and even to publish computational results without even a gesture toward quantification of their numerical accuracy.

N. Metropolis

"...it is an order of magnitude easier to write two sophisticated quadrature routines than to determine which of the two is better."--

J. Lyness at IFIP '71

TABLE OF CONTENTS

<u>Section</u>		<u>Page</u>
1	THE NATURE OF SOFTWARE TESTING	1
	THE ECONOMICS OF SOFTWARE TESTING	1
	THE INADEQUACY OF CURRENT SOFTWARE TESTING PRACTICES	3
	RELATIONSHIP BETWEEN ERRORS AND PROGRAM COMPLEXITY	4
	PURPOSE OF SOFTWARE TESTING	5
	HARDWARE VS. SOFTWARE TESTING	5
	CURRENT APPROACHES TO PROGRAM VALIDATION	8
	Program Proving	9
	Path Testing	10
	Symbolic Testing	11
	A SOFTWARE TEST METHODOLOGY	11
2	PATH TESTING	13
	GRAPH THEORY DEFINITIONS	13
	PROGRAM GRAPHS	17
	Example of a Program Graph	18
	Logic Flow Digraph of a Program	20
	Data Flow Digraph of a Program	21
	PROGRAM GRAPHS AND STRUCTURED PROGRAMMING	22
	THE TREE REPRESENTATION OF A PROGRAM	23
	THE MATRIX REPRESENTATION OF A PROGRAM	25

TABLE OF CONTENTS (Continued)

<u>Section</u>	<u>Page</u>
Reachability	25
Adjacency Matrix	25
Examples	26
LOGIC FLOW ANALYSIS	27
A Cover For The Logic Flow Paths of a Program	28
Methodology	28
Test Effectiveness Measures	29
Example	32
How to Measure Effectiveness of a Given Set of Test Data	32
A Theoretical Upper Bound for The Amount of Testing	34
TEST DATA GENERATION	34
Symbolic Evaluation	35
Linearization	35
Test Case Derivation	35
EXPERIENCE WITH PATH TESTING	36
DATA FLOW ANALYSIS	37
Methodology	38
Dynamic Analysis	39
Static Analysis	40
CONFIDENCE LEVEL OF A TEST	43
Example	45
Specifications for a Test Program	46

TABLE OF CONTENTS (Continued)

<u>Section</u>	<u>Page</u>
ESTIMATING TIME TO COMPLETION	47
Estimating the Number of Fish in a Pond	47
Estimating Number of Errors Not Yet Detected	48
Errors Detected vs. Time	48
Time to Completion	49
3 SOFTWARE TESTING METHODOLOGY AS AN INTEGRAL PART OF SOFTWARE ENGINEERING	50
STRUCTURED PROGRAMMING	51
TOP-DOWN IMPLEMENTATION	51
TOP-DOWN IMPLEMENTATION IN THE CONTINUUM MODEL	52
A COMPREHENSIVE SOFTWARE TEST METHODOLOGY	52
Path Analysis Test Approach	54
Hierarchical Decomposition of the Tree of a Program	54
Program Factoring	54
Optimal Allocation of Resources	55
Path Testing in the Continuum Model	56
4 AUTOMATED TOOLS FOR PATH ANALYSIS	59
STATIC ANALYSIS TOOLS	59
DYNAMIC ANALYSIS TOOLS	60
SQLAB - SOFTWARE QUALITY ASSURANCE LABORATORY	62

TABLE OF CONTENTS (Concluded)

<u>Section</u>	<u>Page</u>
5 APPLICATION TO AIR FORCE PROGRAMS	70
DOD SOFTWARE QUALITY REQUIREMENTS	70
DODD 5000.3	71
SOFTWARE QA AND SOFTWARE TESTING	72
THE ROLE OF PATH TESTING METHODOLOGY IN AIR FORCE PROGRAMS	73
Effectiveness of Test Data	74
An Upper Bound for the Amount of Testing	74
Critical Paths and Modules	74
Specifications for the Test Program	74
Optimal Allocation of Resources	75
Path Testing	75
6 PROBLEMS AND TRENDS FOR THE 80s	76
THEORY	76
METHODOLOGY	77
AUTOMATED TOOLS	78
REFERENCES	80
GLOSSARY OF ACRONYMS	85

Section 1

THE NATURE OF SOFTWARE TESTING

Software is a logical rather than a physical product. The dramatic decrease of hardware costs by a factor of two every two or three years since 1945 has caused the complexity in the logic rather than that in the components to become the principal bottleneck in the development of hardware-software systems. The decade of the seventies emphasized widespread dissemination of and experimentation with software engineering techniques in order to define logical constraints to utilize the richness and complexity of large computer systems made possible in the changing technological environment. Design techniques such as top-down, bottom-up, structural design, programming by operational clusters, hierarchical input-process-output (HIPO) and pseudocode; and programming language characteristics such as data types, data structures, control flow mechanisms, structure of and communication between subprograms have all had significant impact upon the software development process, providing some degree of control and visibility to its development. The art of programming has changed into a scientific discipline in these ten years, rivalling long-established branches of engineering in its breadth and scope and theoretical foundations. Program testing, on the other hand, has remained a collection of not-quite-scientific methods and beliefs that have been likened to a black-art. It is only in the past five or six years that theoretical foundations have been laid for a disciplined approach to assuring quality of programs.

There are two complementary developments in order to extend the scope of software engineering to include quality assurance--verification and validation. The aim of program verification is to establish that computer programs are consistent with detailed specifications. This amounts to program proving. Program validation, on the other hand, attempts to establish empirically the existence of program function and the absence of unwanted function. As a result of the developments in the past six years, program testing based on systematic path analysis of the structure of the program, has become possibly the only effective means to assure quality of a software system of non-trivial complexity.

THE ECONOMICS OF SOFTWARE TESTING

In 1978 the cost of software development, testing and maintenance for the Government was estimated at about \$8 billion,

including \$4 billion for defense applications. The Air Force alone spends about 80-90% of computer systems procurement costs for software, as compared to about 15% in 1955 (46). WWMCCS was estimated to involve .75 billion dollars for software, about 10 times its hardware cost. It is enlightening to note the distribution of this cost in software development stages of program analysis and design, implementation, integration and testing. Fig. 1-1 below shows the cost distribution of six large software projects (1).

Breakdown of Development Costs for Selected Systems

	<u>Analysis and Design</u>	<u>Coding and Debugging</u>	<u>Integrating and Testing</u>
SAGE	39%	14%	47%
NTDS	30%	20%	50%
GEMINI	36%	17%	47%
SATURN V	32%	24%	44%
OS/360	33%	17%	50%
AVERAGE	34%	18%	48%

Fig. 1-1

An average of 47.6% of the total development cost was spent after the code had been developed, in Integration and Testing. The following chart illustrates the relative costs of software over its life cycle for typical large scale programs (47).

Typical Breakdown of Software Costs

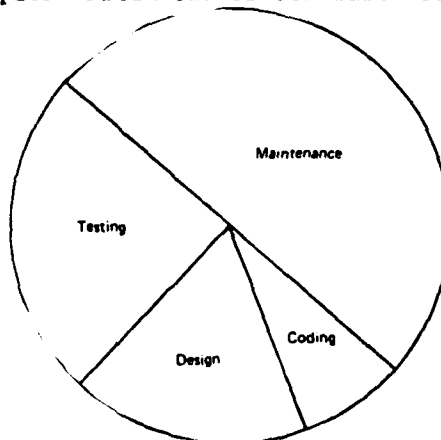


Fig. 1-2

The costs of maintenance and testing account for approximately 75% of the life cycle cost of software. The SAGE system had an average software maintenance cost of approximately 20 million dollars per year after it had been in operation for ten years, compared to an initial development cost of 250 million dollars. In most of the releases of IBM OS/360 operating system, approximately 60-76% of the costs were incurred after the system was made operational (1).

It is clear that about half the total life cycle cost of software systems is spent in generating the software and the other half in making sure that it performs what it is expected to do. Very simply, this says that software actually performs correctly by having been tested into that state, and that testing is not done in a particularly cost effective manner. In fact, as Ed Miller points out, it is quite clear that most of the reliability in current software systems is installed there by hammer and tongs methods, brute force, and "try it and fix it until it works" methods (40).

THE INADEQUACY OF CURRENT SOFTWARE TESTING PRACTICES

The occurrence of a system failure due to software is just as real to the user as when due to hardware. A software error in the on-board computer of the Apollo 8 spacecraft erased part of computer's memory. Eighteen software errors were detected during the ten-day flight of Apollo 14. In the aggregate about \$660 million dollars were spent on software for the Apollo program. Checking of this software was as thorough as the experts knew how to make it. Yet, almost every major fault of the Apollo program, from false alarms to actual mishaps, was the direct result of errors in computer software (61). The U.S. Strategic Air Command's 465L Command System, even after being operational for 12 years, still averages one software failure per day. The rescheduling of the takeoff of the space shuttle Columbia due to software malfunctioning earlier this year is a striking example of the fact that the situation has not improved much.

As the trend towards larger and larger computer systems continues, the consequences of software unreliability become increasingly severe. Software delays during testing often cause delays in a system becoming operational. A six month delay translates into a 100 million dollars in loss of services based upon a projected seven years operational life of a 1.4 billion dollar project (1).

The evolutionary nature of current large-scale software systems, the complexity due to the variety of function, the lag of several years between system requirements definition and system delivery, and the diversity of the operational environment are some of the factors that contribute to the unreliability problem for software.

RELATIONSHIP BETWEEN ERRORS AND PROGRAM COMPLEXITY

Testing costs and program unreliability are increasing functions of the number and type of errors in a program (1). The number and type of errors in a program are related to the complexity of the program. In general, the complexity of an object is a function of the relationships among the components of the object. The complexity of a program design is a function of the relationships among the modules; the complexity of a single module is a function of the connections among the program instructions within the module.

The complexity of poorly structured large systems increases exponentially with their size. To increase the reliability and decrease testing costs, program complexity should be reduced. Two principles are identified from general systems theory to combat complexity (56):

- (i) independence
- (ii) hierarchical structure.

To minimize complexity, maximize the independence of each component of a system. Basically, this involves partitioning the system so that the high frequency dynamics of the system fall within single components and the inter-component interactions represent only the lower frequency dynamics of the system.

A hierarchical structure allows for stratification of a system into levels of understanding. Each level represents a set of aggregate relationships among the parts in the lower levels. The levels may be defined by their functional specifications. The concept of levels allows one to understand a system only to the necessary level of detail. It brings out clearly the interfaces between components at different levels, and allows one to look at the lowest levels of detail within the hierarchy without affecting the top level structure.

In an ideal hierarchically-structured modular system, the complexity at any decision point should be bounded by a constant independent of the size of the system, determined only by the number and complexity of modules immediately affected by the decision.

THE PURPOSE OF SOFTWARE TESTING

Testing is the controlled analysis and execution of a program to validate the pre-specified presence of some program property. The notion of controlled execution forms the basis for a systematic methodology for internal measurement of the behavior of the program. As a minimum, program testing serves to prove the presence of function. When performed systematically, it can also serve to demonstrate the absence of unwanted function.

Although testing can be used both as a post-developmental quality assurance method and as a preventive technique, in this report we will restrict ourselves to testing methodology for programs or modules which have already been coded. In this case, the purpose of software testing is simply to detect errors in the program which is to be tested.

HARDWARE VS. SOFTWARE TESTING

The subject of software reliability has been extensively researched for over twenty years. The reason that there has been no consensus on its characteristics, much less on any of its measures, is that several well-intended Reliability/Availability/-Maintainability (RAM) professionals had an MTTR (mean-time-to-repair) or MTBF (mean-time-between-failure) concept of software reliability. John D. Cooper and Matthew J. Fisher in their book (11) on Software Quality Management, published in 1979, discuss the problem with using traditional hardware RAM techniques for software. Software is different from hardware in that it works the same way every time, it never wears out or deteriorates, spare parts for software are inconsequential. So software reliability is, simply that programs should operate in their operational environment as they are expected to--correctly! It is therefore the confidence in the correct performance of the program that must be established.

In fact, in 1979, Bev. Littlewood (35) formally established that there is something fundamentally wrong in applying the hardware notions of MTTR and MTBF to computer programs, for these concepts may not even exist in general. The argument rests on fairly subtle mathematical points, which have important practical implications.

For hardware there is justification for making the fundamental assumption that the failures are Poisson distributed (3). The time to a failure then follows an exponential law, and the mean time to failure is finite. In such a case the MTBF completely describes the failure behavior of the hardware system. In the case of software, all the failures are inherently present in the system as errors, and there is no deterioration of components, so the distribution of failures as they are executed, may not even have finite moments. In this case MTBF cannot even be assigned a practical meaning. Such a situation is most likely to occur for software for although software errors occur quite rapidly just after it becomes operational, if modifications are not required, the failures become quite sparse, as can be observed in the failure data, from seven programs, depicted below (2). These data were collected for a supervisory program developed for a Government agency by a manufacturer of an operating system. Even though the time between failures may be finite, the expected average of these times after the initial operational phase is quite likely infinite.

Note: Hesse's raw data was in terms of program changes, and the data in this table and the Hesse paper were adjusted by dividing by an estimated 17 changes per bug.

Number of Bugs Removed Per Month for Seven Different Large Programs
(from Hesse (1972) and Shooman (1972)).

	Application A 240,000 Inst.	Application B 240,000 Inst.	Application C 240,000 Inst.	Application D 240,000 Inst.
<u>Month</u>	<u>Bugs</u>	<u>Bugs</u>	<u>Bugs</u>	<u>Bugs</u>
1	514	905	235	331
2	926	376	398	397
3	754	362	297	269
4	662	192	506	296
5	308	70	174	314
6	108	---	55	183
7	---	---	60	158
8	---	---	---	368
9	---	---	---	337
10	---	---	---	249
11	---	---	---	166
12	---	---	---	108
13	---	---	---	31
Total	3,270	1,905	1,725	3,207
Changes				
AVG/ Month	545	381	246	247

	Supervisory A 210,000 Inst.	Supervisory B 240,000 Inst.	Supervisory C 230,000 Inst.
<u>Month</u>	<u>Bugs</u>	<u>Bugs</u>	<u>Bugs</u>
1	110	250	225
2	238	520	287
3	185	430	497
4	425	300	400
5	325	170	180
6	37	120	50
7	5	60	--
8	--	40	--
Total	325	1,890	1,639
Changes			
AVG/ Month	189	236	273

Fig. 1-3

The fundamental assumption of Poisson distributed failures, therefore, does not hold for software. It remains an open question whether any failure law can be developed which reflects the nature of software failures. In the absence of such knowledge, adoption of hardware concepts for software must be avoided.

There seems to be a growing consensus that the characteristics of software reliability as software quality must be designed and built into the structure of the program (11).

A more subtle problem arises in the calculation of availability, i.e., the actual fraction of time the software system will be available. If the behavior of the system is modelled as explained in (34), by an alternating renewal process with the two types of intervals representing operation and repair times, then the observed fraction of a given interval of time (0,T) that the system operates has a finite sample expected value. But the sample expected value may not converge, as $T \rightarrow \infty$, to the actual fraction of time that the system will be available. Of course, if both MTBF and MTTR were finite, then it can be shown that (34, 35)

$$\lim_{T \rightarrow \infty} \frac{\text{Operating time in } (0,T)}{T} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}} .$$

But the left side quantity may converge even if MTBF or MTTR are not finite. It cannot then be guaranteed to converge to the actual fraction of time that the system will be available.

Since, for a given program, it is not possible to prove the finiteness of MTBF, it is meaningless to make any inference about RAM by single numerical measures like sample MTBF. In fact, Bev. Littlewood (35) suggests percentiles of time to next-failure distributions for gaining more insight into the distribution of failures.

CURRENT APPROACHES TO PROGRAM VALIDATION

Prior to the Program Test Methods Symposium in 1972, whatever art of program testing existed was a closely held secret among knowledgeable programmers. But in the past few years several studies and much important theoretical work have produced results which have changed the very nature of software testing. The ad hoc approach of testing software as a black box via a set of trials determined by subjective judgement limited by time and resources is

now being replaced by an integrated, systematic discipline which provides control and visibility into the structure of the program and where scope and reliability are governed by a formal requirement of the specified level of testing and acceptance criteria. Programs are being written which can be used economically to measure the quality of other programs. The fundamental questions of "what to measure?" and "how to measure it in a cost effective way?" are now being answered. There are three basic approaches being taken: Program Proving, Path Testing, and Symbolic Testing.

Program Proving

The program proof process requires the development of a set of verification conditions followed by their detailed analysis and proof. The proof may be done by a mechanical theorem prover.

Program proving methods, at present, have some severe limitations. Proofs can provide assurance of correctness for a program only if the following are true (22).

- 1) There is a complete axiomatization of the entire running environment of the program-language, operating system, and hardware processors.
- 2) The processors are proved consistent with the axiomatization.
- 3) The program is completely and formally implemented in such a way that a proof can be performed or checked mechanically.
- 4) The specifications are correct in that if every program in the system is correct with respect to its specifications, then the entire system performs as desired.

These requirements are far beyond the state of the art of program specification and mechanical theorem proving, and we must be satisfied in practice with informal specifications, axiomatizations, and proofs. Then problems arise when proofs have errors, specifications are incomplete, ambiguous or unformalized. The proof of correctness approach to validation is, therefore, not currently useful except for small combinatorial algorithms. An incorrect program can be proved correct--perfectly properly--relative to an incorrect set of assumptions (22).

Despite the limitations, attempts at program proving are becoming essential to the development of large scale programs which must be understood before being proven correct. Also facilitating provability sets a worthy standard for program structure and specifications.

Path Testing

Much of the theory of path testing arises from two forms of graph theory based modelling of program properties; control flow and data flow. Testing all paths through the graph of a program is usually impractical, even for small programs. Input data about appropriate test forms which will exercise a given flow of the program is inferred directly from the analysis of the internal structure of the program. Path testing seeks to exercise different paths through the graph of a program in a controlled and systematic way in order to develop metrics for the effectiveness of the test program. Path testing is usually undertaken to achieve certain goals of program coverage like all statements or all branches tested at least once.

Studies in the reliability of path testing are of theoretical as well as practical interest because they provide an upper bound on the reliability of strategies that call for testing of a subset of program paths.

Program faults which cause a program with an error to differ from one that is perfect can be put into two categories: case errors and action errors. A case error exists when there is a fault in a program's decisional structure which causes it to differ from the correct program in a way that drastically changes the implied partitions of the input space, i.e., the so-called program "cases." An action error exists: (a) in the absence of a case error and (b) when a wrong output would be produced when the program is executed with valid test data for that case.

The pioneering work of Goodenough and Gerhart, and subsequent work of Howden in the past six years has established that for computer programs which satisfy certain structural constraints, program testing is the full equivalent of a proof of correctness. Reliable tests can be derived for the rather wide class of programs that contain no case errors. The general character of reliable path testing seems to suggest the likelihood of extending the classes of program faults against which testing would be effective, although this may require the use of special programming devices in some cases.

As a result of these developments, path testing has become possibly the only effective method to ensure the quality of a software system of non-trivial complexity. But many of the techniques that are used need serious further development--both theoretical and empirical, through accumulation of experiences in practical applications.

The goal of path testing is the same as that of program proving: to guarantee the absence of errors in a computer program. Whereas program proving is a reductive process, path testing is inherently a constructive process since every new path tested contributes information about the quality of the program being tested. Path testing has the advantage of providing accurate information about a program's actual behavior in its actual environment, so errors found during testing, like infinite loops or division by zero, can be corrected by human intervention. A proof is limited to conclusions about behavior in a postulated environment. Testing and proving are complementary methods for decreasing the likelihood of program failure.

Symbolic Testing

Symbolic testing systems are a rather recent innovation that combine features of path analysis and a limited form of program interpretation. Instead of dealing with actual input values, a symbolic testing system acts on the formulas that result from considering the tree of possible program flows that begin at the invocation point in a program. The tree is pruned as much as possible; that is, infeasible paths based on data constraints are removed as soon as they are discovered to be infeasible. In this way the growth of the tree is kept within reasonable limits and the system can handle practical sized programs. There is a close relationship between the operation of a symbolic testing system and a program prover: symbolic analyses simulate "execution" of a program path that has rather thoroughly known properties. It is expected that symbolic testing techniques will grow in importance in future years, potentially to a point where they form one of the main tools supporting the program testing activity.

A SOFTWARE TEST METHODOLOGY

The framework for a formal theory of software testing was developed by Dijkstra. In 1975, Goodenough and Gerhart proved the fundamental theorem establishing that programs which meet certain

structural criteria can be proved correct by program path testing (22). This theory, extended by Howden (27,28) has brought software testing into the domain of software engineering. For top-down structured programs, path analysis techniques have been developed which utilize the hierarchical structure of the program, providing control and visibility into its structure.

In this report a quantitative methodology, based on path testing, will be developed, as a natural extension of software engineering techniques to the testing phase of software development. Test metrics are developed to provide control and visibility into the structure of the program. Test objectives are quantified in terms of program path coverage, and quantitative acceptance criteria are developed. Automated tools for path analysis are discussed. Application to Air Force programs is discussed in Section 5.

An application of this methodology was made to the Automatic Speaker Verification Algorithm for the BISS/ECS (Base and Installation Security System/Entry Control System). Details of this application are being written up as a separate working paper.

Section 2

PATH ANALYSIS

In the path analysis testing strategy different paths through a program are analyzed which exercise the control structure and the data flow through the program. The program is modeled as a graph consisting of a set of program flow paths which, if executed by a set of test data T, would reveal an error along a path if the program was incorrect.

The execution-time theory of software testing advocated by John D. Musa (45), where a program is made to execute in its natural environment for a specified amount of time determined only by the available resources, or similar theories, treat a computer program as a black box and ignore the structure of the program. As a result, the most commonly traversed paths through the program get tested over and over again, providing no additional information about the correctness of the program and wasting valuable time and other resources. On the other hand, several of the less frequently traversed paths do not have much of a chance of being tested. When the program becomes operational and a less frequently traversed path is encountered, the program may contain an error along this path, and despite considerable expense in time and resources, the program is likely to cause a system failure. A more enlightened "white-box" approach models the logic and data flow in a program using graph theory techniques. The objective of the graph theory approach to analyzing programs is to infer data about appropriate test forms directly from the internal structure of the program. This model allows relatively simple measurements of the thoroughness of testing accomplished up to any point during the test program. It provides a visibility into the program so that appropriate analysis can be carried out to allocate resources optimally and to eliminate any redundant testing. Also test effectiveness measures can be developed so that the amount of testing required for a desired degree of confidence in the correctness of the program can be determined. These measures are sensitive to cost and software criticality factors. Such an approach to software testing will now be developed.

GRAPH THEORY DEFINITIONS

The technology of program testing is strongly rooted in the notion of directed graphs. A directed graph or digraph consists of a set of nodes and edges that are oriented to indicate flow from the

originating node to the terminating node. For a graph G consisting of nodes V_1, \dots, V_n and edges E_{ij} for (i,j) in some subset of $\{1, \dots, N\} \times \{1, \dots, N\}$, the following definitions are made:

Predecessor and Successor Nodes. A node V_i is a predecessor of node V_{i+1} if there is a directed edge E from V_i to V_{i+1} . V_{i+1} is then called a successor of V_i .

Indegree and Outdegree of Nodes. The number of incoming edges to V_i is its indegree. The number of outgoing edges from V_i is its outdegree.

Entrance and Exit Nodes. A node V_i is an entrance node for the graph G if V_i has no predecessors in G . V_i is an exit node if V_i has no successors in G .

Decision Node. Any node which has more than one outgoing edge is called a decision node. The entry and exit nodes of a digraph are also decision nodes.

Subgraph. A portion of a graph containing a subset of edges and nodes of the graph and which is itself a graph, is called a subgraph.

Reduced Graph. When each node in a graph which has outdegree one, except the entry node, is identified with its successor node and the edge between them is dropped, then the graph is said to be reduced.

Path. A sequence of nodes

$$V_{i1}, V_{i2}, \dots, V_{in}$$

is called a path if each V_{ik} has the property that V_{ik-1} is a predecessor node of V_{ik} and V_{ik+1} is a successor node of V_{ik} . V_{i1} is called the entry node for the path and V_{in} is called the exit node. If the entry and exit nodes of a path coincide with the entry and exit nodes of the graph G , then the path is called a complete path.

Length of a Path. The length of the path

$$V_{i1}, V_{i2}, \dots, V_{in}$$

is defined to be $n-1$.

DD Path. A path in a reduced graph is called a

Decision-to-Decision path or a DD path.

Segment. A DD path of length one is called a segment.

Cycles. If the entry and exit nodes of a path coincide, then the path is called a cycle or a loop.

A cycle is called an (m,n) cycle if its entry node has indegree m and outdegree n .

Connected Graph. A graph is connected if there is at least one path between every pair of nodes.

Tree. A digraph is called a tree if it has a single entry node and each node except the entry node has indegree one.

Subtree. A subgraph of a tree which is also a tree is called a subtree.

Leaf. A subtree consisting of only two nodes one of which is an exit node is called a leaf.

The above definitions allow one to describe a program's structure, decompose a program into simpler substructures, determine whether there exists unreachable code, and to develop metrics for the effectiveness of a software testing strategy.

These definitions are illustrated below.

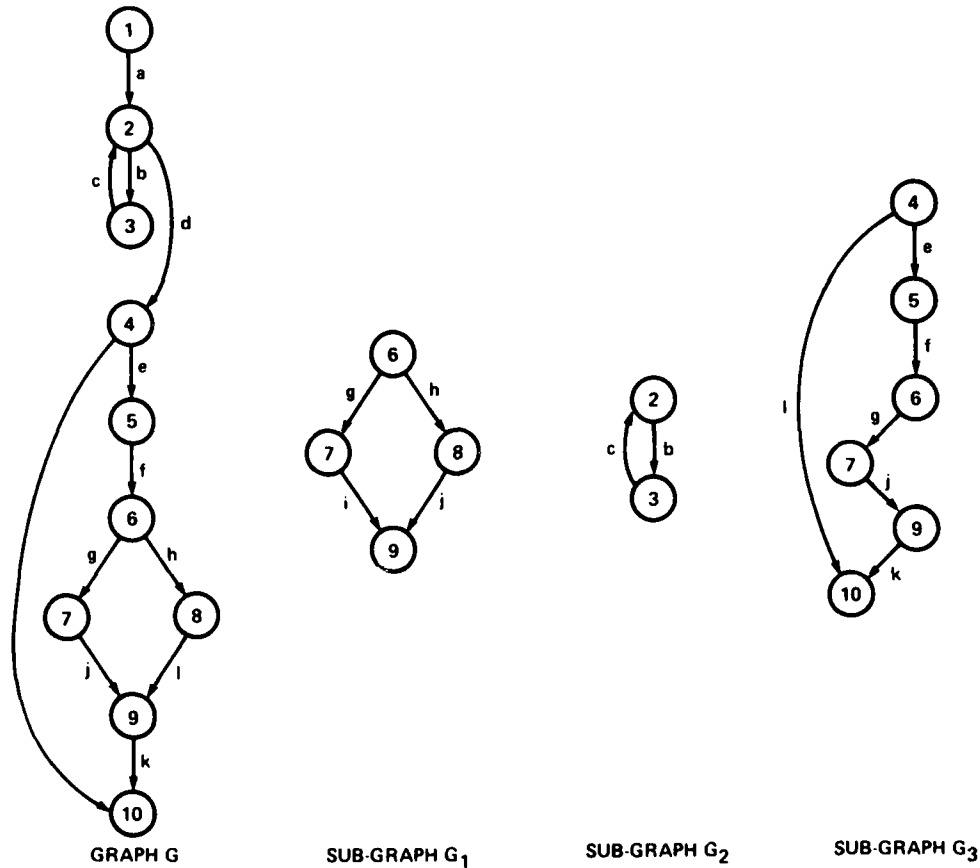


Fig. 2-1

The digraph G given above has 10 nodes (1). The edges of this digraph are shown in the illustration as a, b, ..., k, l. Graph G and subgraphs G_1 and G_3 are all disconnected graphs. Note that there is no path between nodes 7 and 8 in G and G_1 . G_2 is connected. Nodes 1, 2, 4, 6 and 10 are decision nodes. In order to reduce the Graph G, nodes 3, 5, 7, 8 and 9 should be removed and the following edges should be merged:

b and c and d
e and f
g and i and k
h and j.

The following graph T is a tree. A tree does not contain any cycles.

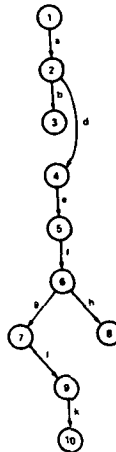


Fig. 2-2

PROGRAM GRAPHS

A program is an ordered set of all its executable statements. A program may be represented by a graph in the following way: the nodes represent statements within a program which the execution-point could pass through and the edges represent the actions which the program takes in getting from one node to another. The beginning of the program corresponds to the entry node in the digraph, and the exit statement, after the invocation of the program is complete, is the exit node. The entry node has no incoming edges and the exit node has no outgoing edges. Any node which has more than one outgoing edge is a decision node and corresponds to a decision statement in the program. The sequential execution of statements in a program creates nodes which have only one incoming edge and one outgoing edge. Such a node and the two edges involved can be merged into one edge which represents the combined action of both the statements in the reduced graph. In its reduced form, the digraph effectively represents the decision of the program, with each possible decision outcome assigned to an outgoing edge of a node. The program is thus reduced from its original source-text form to a set of DD paths. The segments of this program are simply blocks of statements which must always be executed together as the result of some decision taken by the program at a decision node.

The digraph representation of a program is useful in analyzing the macroscopic structure of programs, especially in developing test cases and associated test data. In particular, this type of analysis would be useful in constructing new test data to exercise a previously unexercised segment in a complex program.

Example of a Program Graph

Consider the following program (52):

```

1  SUBROUTINE BUBBLE (A,N)
2      BEGIN
3      FOR I = 2 STEPS 1 UNTIL N DO
4          BEGIN
5              IF A(I) GE A(I-1) THEN GOTO NEXT
6              J = I
7          LOOP: IF J LE 1 THEN GOTO NEXT
8              IF A(J) GE A(J-1) THEN GOTO NEXT
9              TEMP = A(J-1)
10             A(J) = A(J-1)
11             A(J-1) = TEMP
12             J = J - 1
13             GOTO LOOP
14         NEXT: NULL
15     END
16     END

```

Label the edges in the flow of the program as follows:

Label	Edge	Label	Edge
a	(1,2)	m	(8,14)
b	(2,3)	n	(8,9)
c	(3,4)	p	(9,10)
d	(3,16)	q	(10,11)
e	(4,5)	r	(11,12)
f	(5,14)	s	(12,13)
g	(5,6)	t	(13,7)
h	(6,7)	u	(14,15)
j	(7,14)	w	(15,3)
k	(7,8)		

Fig. 2-3

The digraph of this program is given below:

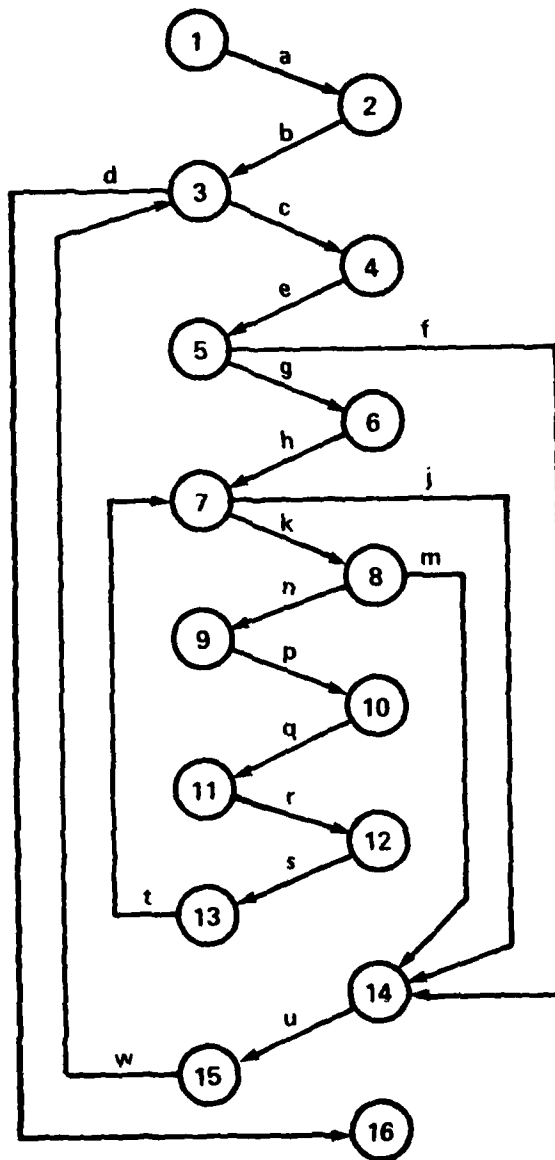


Fig. 2-4

The digraph can now be reduced as follows:

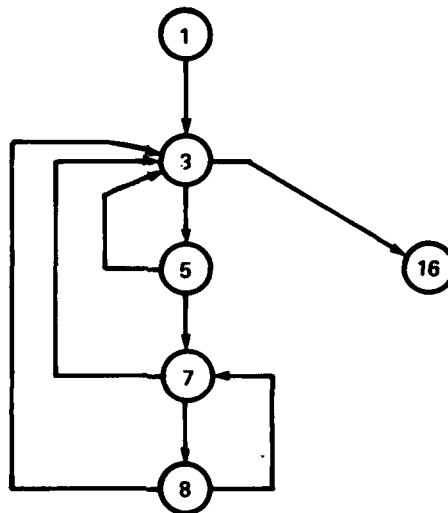


Fig. 2-5

Nodes 1, 3, 5, 7, 8, and 16 are decision nodes for this program. The edges represent logical sequences of statements which must be executed together. The digraph captures the inherent logical structure of the program. The DD paths are obvious, those of length one are given below.

(1/3)	= a b
(3/16)	= d
(3/5)	= c e
(5/7)	= g h
(5/3)	= f u w
(7/8)	= k
(7/3)	= j u w
(8/7)	= n p q r s t
(8/3)	= m u w

Logic Flow Digraph of a Program

The logic flow digraph of a program can easily be derived from the program by examining the decisional statements within the program. In FORTRAN for example, these are IF, GO TO (...), and DO statements, and some others that involve statement labels and conditional transfers. In COBOL, the decisional statements are IF PERFORM and several other statement types.

The graph G of Fig. 2-1 is the logic flow digraph of the following ALGOL procedure (55):

```

PROCEDURE TEST CONDITIONS:
COMMENT TEST ALL CONDITIONS FOR MEMBER IDENTIFIED BY CURRENT NODE;
COMMENT IF ALL CONDITIONS HOLD ADD MEMBER TO LINKED LIST;
  BEGIN
    INTEGER A, I;
    FAIR:=TRUE;
    I:=1;
    WHILE ((REQUEST (I) ="Q") AND (FAIR = TRUE)) DO
      BEGIN
        FAIR:=MATCHINE(I);
        I:=I+1;
      END;
    IF FAIR = TRUE THEN
      BEGIN
        A:=ALLOCATE1;
        IF LIST POINTER = NIL THEN LIST POINTER:=A
        ELSE SETCDR1(LAST,A);
        LAST:=A
        SETCDR1(LAST,NIL);
        SETCAR1(LAS,CDR2(CURRENT NODE+1));
      END;
    END TEST CONDITIONS;

```

The circled nodes 1, 2, 4, 6, and 10 are decision nodes and the corresponding digraph can now easily be generated. The three constructs If Then Else, While Do, and If Then are represented by the three subgraphs G_1 , G_2 and G_3 of Fig. 2-1.

DATA FLOW DIGRAPH OF A PROGRAM

Another graph form that is useful is the data flow graph, which models the dependence between variables in the program on each other and on external variables that are used as the input and output for the program. In a data flow graph each node corresponds to a variable, and the edges indicate the dependence between variables. When the program computes the variable A from the content of the variable B, for example, there is an edge from the node B to the node A. Similarly, when the result of generating B is used in the final output of the program C, there would be an edge going from B to C.

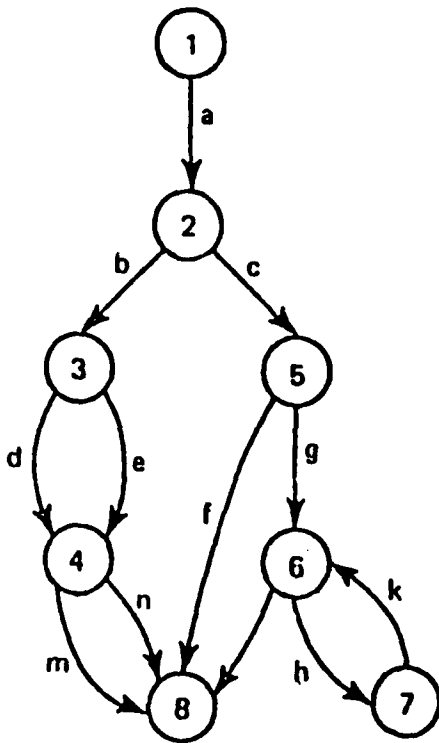
The data flow graph makes it possible to determine some important properties such as the interdependence between segments.

Similarly, the data flow graph can be used to "prove" the allegation that all variables are set before they are used in the global, or multiple module, sense.

PROGRAM GRAPHS AND STRUCTURED PROGRAMMING

When a program iterates, the digraph of the control structure will have cycles. A single entry single exit cycle is the kind that arises in a purely structured program through the use of the WHILE...END WHILE construct. Following are examples of digraphs of a structured and an unstructured program. Cycle (3,2,3) of the unstructured program is a (2,3) cycle.

STRUCTURED PROGRAM GRAPH



UNSTRUCTURED PROGRAM GRAPH

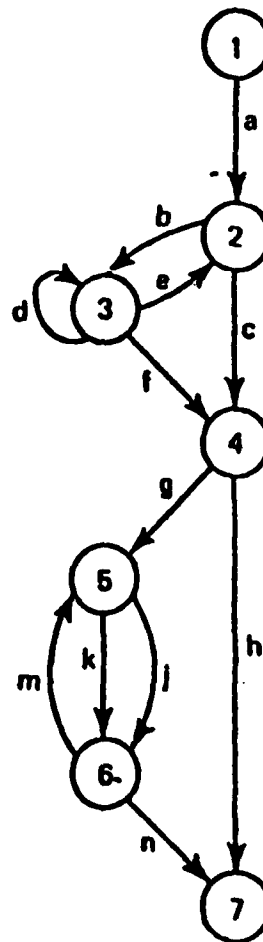


Fig. 2-6

Multiple entry and exit cycle structures complicate the path analysis of a program. Fortunately any unstructured program can be automatically structured by the following two-step technique (40).

(1) Each (m,n) cycle is copied over m times to result in a set of m different (1,n) cycles.

(2) Each (1,n)-cycle is then broken down into a (1,1)-cycle and a (1,n-1)-cycle, when n is greater than 1. Each (1,1)-cycle corresponds to an iteration, and the remaining cycles are decomposed in turn until nothing other than (1,1)-cycles remain.

All programs can therefore be represented in purely structured form, using only succession, alteration and iteration primitives (IF...ELSE...END IF, WHILE...END WHILE). This representation of a program may, of course, require additional variables and edges to be defined. In this document, we will assume that the program to be tested has been represented in a purely structured form.

THE TREE REPRESENTATION OF A PROGRAM

The logical structure of a perfectly structured program can be uniquely represented by a tree. Such a program is constructed with three programming primitives: succession, selection, and iteration. These are denoted by ., +, and *, respectively. The nodes of the tree correspond to the three primitives ., +, and * and a directed edge emanating from a node V denotes the sequence of non-decisional statements which must be executed as a consequence of the decision made at V. The following conventions are made here:

During an execution of the program, succession (.) gives control of the program to the leftmost successor node which has not been traversed so far. Hence every succession (.) node is traversed exactly twice because when a leaf is reached after the first traversal then the program backs up to the last succession (.) node.

The left edge emanating from a selection (+) node corresponds to the true outcome and the right edge corresponds to the false outcome.

The left edge emanating from an iteration (*) node corresponds to repeated action or loops, and the right hand descendant corresponds to an exit condition.

Consider the following program (40). The number in the parentheses indicates the number of statements in the nondecisional sequences A,B,C,D,E, and F.

```

IF P1
  A(4)
ELSE
  IF P2
    B(4)
  ELSE
    C(2)
  END IF
  WHILE P3
    D(2)
  ENDWHILE
  IF P4
    E(6)
  ELSE
    F(1)
  END IF
END IF

```

The logical structure of this program is given by the following tree:

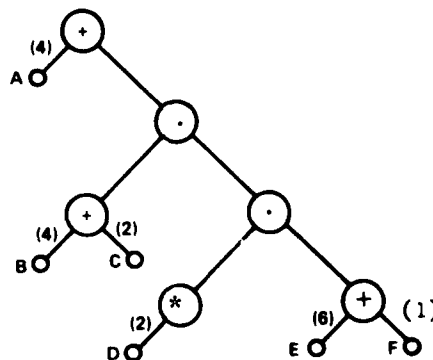


Fig. 2-7

The exit nodes of the leaves of this tree correspond to END or END IF statements. Note that the non-decisional statement sequences always show up as leaves in the tree.

This representation of a program is unique in the sense that any two programs that have the same internal organization of control statements will have precisely the same tree.

THE MATRIX REPRESENTATION OF A PROGRAM

Adjacency Matrix

The adjacency matrix A of a program is a matrix representation of the reduced graph G of the program. The adjacency matrix of graph G which has N nodes is the $N \times N$ matrix defined as follows:

$$\begin{aligned} A(i,j) &= 1 \text{ if there is a directed edge from } i \text{ to } j \\ &0 \text{ otherwise,} \\ 1 \leq i, j \leq N. \end{aligned}$$

The Adjacency matrix and its powers provide program path information which can be used to identify the paths whose correct execution should be verified. This is illustrated in the examples given on the next page.

Reachability Matrix

If a program has E executable statements then the reachability matrix R of the program is defined to be the $E \times E$ matrix whose (i,j) th element is

$$\begin{aligned} R(i,j) &= 1 \text{ if the digraph of the program admits a directed path} \\ &\text{between node } i \text{ and node } j. \\ &0 \text{ otherwise,} \\ 1 \leq i, j \leq E. \end{aligned}$$

The reachability matrix can be used to ascertain whether any program code is not used. This is indicated by one or more zero columns in R (except for the column that corresponds to the entry statement). For a large program with several modules, the generation of the reachability matrix and detection of unreachable code can be done by automated tools. Some of these tools--RXVP, SQLAB, JAVS are described in Section 4 of this document.

The reachability matrix can be used to identify and test the ways in which a given node can be reached. The reachability matrix of the graph G of Fig. 2-1 is given in the next example.

Reachability may alternately be defined for each node j which represents an executable statement in the unreduced digraph of a program as the number of distinct nodes from which node j can be reached.

If $r(j)$ denotes the reachability for node j then

$$r(j) = \sum_{\text{all nodes } i \neq j} R(i, j).$$

Average reachability for j is defined as

$$r(j)/E.$$

The reachability for a node j is somehow related to its complexity. A high reachability for a node therefore indicates the relative importance of this node and the edges emanating from it for the correct execution of the program. Hence, such a node should be accorded corresponding emphasis during testing.

Examples

The Adjacency Matrix of the program G of Fig. 2-1 is given below.

	1	2	3	4	5	6	7	8	9	10
1	0	1	0	0	0	0	0	0	0	0
2	0	0	1	1	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	1
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	1	0	0
7	0	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0

The Reachability Matrix R of G is given below.

	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	1	1	1	1	1	1
2	0	1	1	1	1	1	1	1	1	1
3	0	1	1	1	1	1	1	1	1	1
4	0	0	0	0	1	1	1	1	1	1
5	0	0	0	0	0	1	1	1	1	1
6	0	0	0	0	0	0	1	1	1	1
7	0	0	0	0	0	0	0	0	1	1
8	0	0	0	0	0	0	0	0	1	1
9	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0

LOGIC FLOW ANALYSIS

The purpose of the digraph representation of a program is to make visible and reachable the internal structure of the program in order to derive a set of tests which are effective in verifying that the intended software functions of the program are present. Even if at all possible, it is not practical to test all DD paths through even relatively simple programs. The flow of a small program is represented in the following reduced digraph. The program consists of a DO loop which is executed anywhere from 0 to 10 times, followed by a two-way IF, followed by another DO loop which can be executed anywhere from 0 to 10 times. Each loop contains a set of nested IF statements.

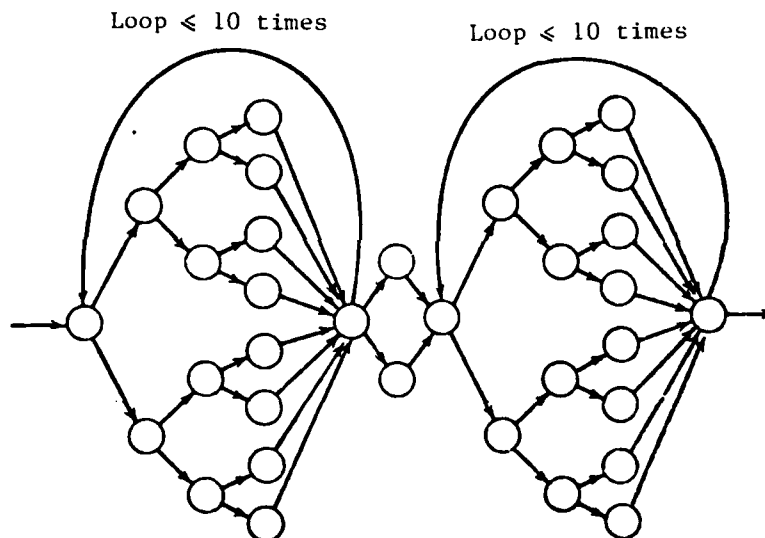


Fig. 2-8

Under the worst case assumption that each decision node is traversed independent of others, there are approximately 10^{18} distinct DD paths through this program. The estimated age of the universe, in comparison, is only 4×10^{17} seconds. Of course, in practice all these paths are not independent and as such some of these may not be considered for testing. But even the number of feasible paths through a program is usually very large. Executing all feasible paths through the TITAN missile navigation and guidance software would take an estimated 60,000 hours of CPU time.

For effective program testing, therefore, a set of test cases must be carefully chosen which will provide the optimal amount of confidence in the correctness of the program given a certain amount of resources. The rest of this section will deal with the development of such a methodology for testing.

A COVER FOR THE LOGIC FLOW PATHS OF A PROGRAM

An immediate simplification of optimally choosing test cases for a program is obtained by the boundary-interior method for path testing (28). In this method, two paths are defined to be equivalent if the only difference between them is the subpath they follow through one or more loops during some traversal of the loop other than the first traversal.

A program is thus decomposed into a finite set of equivalence classes of paths in such a way that an intuitively complete set of test cases would cause the execution of one path in each class. A set of complete paths through the program which contains at least one element from each equivalence class is called a cover for the program.

In the path analysis approach to testing, the programmer utilizes the knowledge of the internal structure of the program as represented by its DD paths in order to construct an ever increasing sequence of test cases. A test consists of an execution of the program along a chosen path. The output of the test is compared with that which was supposed to be generated for the given test data. This approach to testing makes it possible to develop increasing levels of confidence in the correctness of a program. At any time during the test program, test effectiveness measures can be developed, as will be shown later in this document. If the amount of confidence is not satisfactory, then in order that different parts of a program's capability can be tested, path analysis techniques can identify those segments which have not been tested so far. The average reachability of the decision nodes from which a segment emanates can help identify the most significant segments for which tests must be generated. Whereas an ultimate software testing system would include automatic test data generation for a given segment and an automatic analysis of the outputs, at present there is only limited capability to do so, although these are subjects of much current research (7,9,12,23 and 25).

Typically, a cover is made up of paths through the program which traverse the loops in the program exactly 0 or 1 times.

Thus, the following three paths form a cover for the program of Fig. 2-9;
{1,2,4}, {1,2,3,2,4}, {1,2,3,3,2,4}.

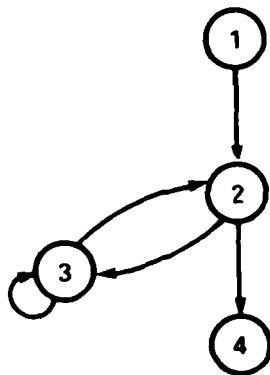


Fig 2-9

The goal of path testing is to ensure that a sufficient number of statements, DD paths and subroutine calls are exercised during program execution, with the least amount of redundancy, in order to achieve a certain level of confidence in the correctness of the program. Test effectiveness measures, which will be developed in this section, will be the tools to assess sufficient coverage of statements, branches and DD paths for a given test program. It is well accepted that an effective test program should at the minimum contain test data to execute each statement and each branch at least once. In fact, the Air Force is considering the adoption of this minimum testing standard for all its programs (39, p. 404).

In Section 3 path testing will be presented as a natural extension of currently practiced software engineering practices. Along with structured programming and top down modular implementation, path analysis will be shown to form an integral part of software engineering. It will also be shown that path testing can be utilized in the continuum model of software development which models the evolutionary development of structured computer programs in a top down fashion.

Test Effectiveness Measures

Having chosen a path analysis approach to software testing, one is faced with the problem of how thoroughly should the program be tested. Testing with real data until resources run out, as it is usually done, does not provide the test leader with an estimate of the effectiveness of the test program. Although there are not many formal test results which provide guidelines concerning the optimal amounts of path

testing given for a program, it is generally agreed that every statement and every branch of the program should be exercised at least once. However, in Howden's experiment (62) branch testing by itself exposed only 21% of the errors, and lead him to conclude that "detection of a significant number of errors that will be discovered by path testing depends on the combinations of program branches rather than single branches."

In this section we will develop a hierarchy of test effectiveness metrics which will quantify with increasing confidence, the level of test effectiveness achieved. Also the scheme will help identify physical areas in the program which have not been covered so far in the test process. Define the following test effectiveness metrics:

$$TEM(0) = \frac{\text{number of distinct statements exercised at least once}}{\text{total number of executable statements}}$$

number of distinct DD paths of length no more than k exercised at least once

$$TEM(k) = \frac{\text{number of distinct DD paths of length no more than k exercised at least once}}{\text{total number of DD path of length no more than k}}$$

for k = 1, 2,

A DD path of length k is a program path through k+1 decision nodes in the program. It is assumed here that paths are considered equivalent if they are different only in the subpaths they follow through the loops during traversal of the loop other than the first traversal. In this case a path which traverses the loop a minimum number of times (at most once) may be chosen as a representative path.

If there are no cycles in the program graph then the hierarchy of test effectiveness measures will be terminated at k = m where m is the length of the longest path through the program.

The program will then be completely tested if

$$TEM(m) = 1.$$

Note that if there are no unreachable statements then

$$TEM(m) = 1 \Rightarrow TEM(m-1) = 1 \Rightarrow \dots \Rightarrow TEM(k) = 1 \Rightarrow \dots \Rightarrow TEM(1) = 1.$$

Different TEMs measure structural test effectiveness at different levels. TEM(0) = 1 corresponds to executing all statements at least once. TEM(1)=1 corresponds to testing all segments at least once. Note that a non-executable statement may not belong to any segment. TEM(2) = 1 corresponds to the well known branch testing technique, i.e., the construction of test cases to exercise all branches in a program at least once. A test requirement of TEM(2)=1 is the first intermediate level of test effectiveness in the sense that it forces the tester to choose paths through the program which capture some of the structure of the program. Attempting to achieve unity at progressively higher levels in the hierarchy provides a more formal and systematic approach to software testing than relying on the programmer's intuition or the "black box" approach of execution time-testing. When TEM(k) is maximized for some k, then DD paths of length k+1 are examined. Some of these might already have been tested by data which were used to exercise length k paths. Those that have not yet been executed may be identified and an attempt made to generate data for their execution. This would yield sets of test paths as independent of each other as possible so that different parts of a program's capability can be addressed.

The above systematic approach to software testing, although practical, is not always easy to use and therefore must be augmented by other testing techniques.

1. Infeasible Paths: Any automatic path generation technique for deciding on which paths to test leads to the problem of paths which can never be executed. A certain program action taken at one point in a program may result in a set of conditions that makes some other program action impossible. Woodward et. al (62) point out that although the number of paths through a program component rises dramatically as paths of ever increasing lengths are considered, very large proportions of them are infeasible (62). The problem must be handled by augmenting the current automatic tools with a certain amount of manual analysis.
2. Infinite Loops: The above testing scheme provides less than adequate testing for infinite loops in the program. In practice, however, it is easy enough to determine a large enough k for a given program such that the existence of a feasible DD path of length k would indicate an infinite loop in the program.

Example

Consider the following program:

```
1  SUBROUTINE EXAMPLE(N,RESULT)
2  NSUMSQ =0
3  DO 10 I=1,N
4    NSUMSQ = NSUMSQ + I*I
5 10 CONTINUE
6  RESULT = SQRT(FLOAT(NSUMSQ))
7  RETURN
8  END
```

The reduced digraph of this program is given below



Fig. 2-10

The DD paths of level $k \leq 3$ are:

(1,3) (3,5) (5,3) (5,7)
(1,3,5) (3,5,3) (5,3,5) (3,5,7)
(1,3,5,7) (1,3,5,3) (3,5,3,5) (5,3,5,7) (3,5,3,7)

Testing the following two paths through the program will satisfy the condition $TEM(3)=1$;

(1,3,5,7,exit)
(1,3,5,3,5,7,exit).

These paths correspond to inputs $N=1$ and $N=2$ for the program.

However these two paths do not test for infinite loops through the program.

How to Measure Effectiveness of a Given Set of Test Data

Suppose one wishes to evaluate the effectiveness of a given set of test data D for a program. This may come from real life experiments that were conducted for a previous program, or some data

or pseudodata generated during the developmental stages of the program. The effectiveness of this data may be determined by developing metrics for this data. Define

$$\text{DEM}(0) = \frac{\text{number of distinct statements executed by the data set } D}{\text{total number of executable statements}}$$

$$\text{DEM}(k) = \frac{\text{number of distinct DD paths of length less than or equal to } k \text{ executed by this data}}{\text{total number of distinct DD paths of exact length } k}$$

$$k \geq 1.$$

For $k=0,2$, this would immediately indicate how many statements and branches had not been executed by this data. Higher values of k correspond to combinations of branches that are executed by this data. Values of $\text{DEM}(k)$ close to one give high degree of confidence in the set of experimental data. The effectiveness of any set of test data can be measured at different levels k by the values of $\text{DEM}(k)$.

Measuring $\text{DEM}(k)$ for a program can easily be automated to some extent. What is needed is a mechanism for recording whether or not a program's flow-of-control passes through an action which results from a particular value of a predicate outcome. So, the program need just be instrumented, using the given data, in such a way that each of the decisions of the program is recorded in some manner. Then the recorded data, called a decisional trace, can be analyzed to determine the values of $\text{DEM}(k)$ for a given value of k , which are achieved by this test. The aggregate results of these computations for the entire data set measure the effectiveness of the testing activity with this data set. This measure may be applied to a module of a program, or to the entire program.

An outstanding feature of using path testing as opposed to any other software metrics in measuring the effectiveness of a data set in the field is that the Data Effectiveness Measures increase in value only when a new test performs something functionally different from what has been tested before. Merely executing the same paths through a program which donot add information about the correctness of the program, do not increase the value of $\text{DEM}(K)$.

A Theoretical Upper Bound for the Amount of Testing

How much testing is enough for testing a computer program thoroughly? An upper bound can be determined on the number of paths that must be chosen properly and executed in order to cover every branch in the program. This upper bound, called the cyclomatic number or the Paige-Holthouse measure, is related to the complexity of the program(7,25). In practice, this number of appropriately chosen paths through a program will not establish the correctness or incorrectness of the program. The practical use of computing the cyclomatic number lies in the guidance it provides in comparing the relative complexity of different modules of a program in order to optimally allocate the available resources for testing.

For a reduced program graph G with E edges and V nodes, an upper bound for the number of tests for total edge coverage is

$$T(G) = E - V + 2.$$

This is easy to see because if a test reaches any decision node (except the exit node) then it must exit somewhere. So at least one of the edges coming out of the decision node will be automatically covered. There are (V-2) decision points in G. Hence E-(V-2) paths can be chosen which will cover every edge in G.

The Air Force is considering adopting k=2 level of testing, i.e., branch testing as a minimum standard for testing software programs(39, p. 404). The cyclomatic number can then be used to determine how far the test program has met the goals at a particular point.

TEST DATA GENERATION

The test data generation problem is to find an algorithm which, given any class of paths, will either generate test data that causes some path in that class to be executed or determines that no such data exists. This problem is theoretically unsolvable(27). Even the problem of construction of certain constrained paths through a program has been shown to be NP-complete(19). This, however, does not mean that non-algorithmic or heuristic techniques cannot be used.

There are two parts to the test data generation problem. First, given a DD-path to find another one which can potentially lead from an invocation of the program through this DD-path; and secondly having determined the program path from entry to exit, to determine input data which will cause the program to execute along this path. Having identified a particular program path to be

executed, there will be a set of predicates that have particular values along that path. By backtracking along this path, the general problem of test data generation can be reduced to one of solving a set of simultaneous inequalities involving program variables that define the set of conditions necessary for a particular program flow to actually occur. Typically, these inequalities are non-linear, which makes the problem very difficult. There are three basic approaches to the solution which have attained partial success:

Symbolic Evaluation

This method involves either forward or backward symbolic interpretation of actual program statements that lie along a chosen path. Research in this area centers on selecting the particular statements to be included in the analysis and the ways to process the resulting formulas. IBM (33) and Stanford Research Institute (4) are actively seeking to exploit the similarities between symbolic evaluation and path analyses methods for automatic generation of program data. This method at present is only for theoretical interest. The practical applications will probably come in another decade. In 1978 Howden (26) attempted to use symbolic evaluation on six different programs and concluded the method was of little practical use at present.

Linearization

In this method of test data generation the inequalities describing the conditions of execution for a program path are derived. In all but the most trivial cases, simultaneous nonlinear inequalities must be solved. One partially successful technique for solution involves linearizing the inequalities and solving the linear system in order to approximate a solution of the nonlinear system. If the solution to the linear system does not solve the non-linear system then the linearization process is continued another time for a more appropriate solution. The method has been automated (4), but it does not guarantee an exact solution in a given time.

Test Case Derivation

Test data sets can be derived by altering an existing set of test data so that the altered data set forces the program to execute a previously unexecuted segment. The method, described in (51) is based on a series of heuristically guided searches or variations of a known data set which execute the program "near" an unexercised segment.

Semi-automated assistance is provided by program analysis tools for the test data generation problem. These tools help in identifying structurally feasible program paths which contain a given DD path and in performing some of the computations needed in the analysis of that path. More sophisticated assistance which is based on an analysis of the collection of paths which are "close" to the desired path is provided by tools like the NASA ATDG system. Although this and several other tools can be used as an aid to generate test data, as Ed Miller puts it, "there have been several interesting proposals, but very little automatically generated data."

EXPERIENCE WITH PATH TESTING

The effectiveness of a testing program should be measured by how many errors it can catch. At the current state of development, the path testing methodology needs to be augmented by other types of testing techniques, although it is of invaluable help in determining which areas of the program in which to concentrate for better coverage.

It is possible to execute all control flow paths through a program without detecting missing control flow path errors. This type of errors arise from failure to test for a particular condition, resulting in inappropriate action. For example, failure to test for a zero divisor before executing a division may be a missing control flow path error. Of course, a detailed data flow path analysis would detect this particular error.

Practical experiences with path testing are reported below.

D. S. Alberts (1) reports that the use of test coverage measures caught between 67 and 100 percent of the errors and at 2-5 months earlier than they would otherwise have been detected. The particular automated tools used applied branch testing. J. R. Brown reports (7) that using branch testing eliminated nearly 90% of the program's errors. It wasn't clear, however, whether this resulted simply from requiring the programmers to examine their code very carefully. W. E. Howden in 1978 reported on an experiment using six sample programs. Branch testing by itself reliably exposed only six out of 28 errors (about 21%). The path testing strategy itself was reliable for exposing 18 out of 28 errors (64%) and was the best single testing strategy at exposing errors. This indicates that detection of a significant number of errors depends on the combinations of DD paths rather than single branches.

DATA FLOW ANALYSIS

From the point of view of control theory, the action of a software program can be captured by rigorously defining all its intermediate states. In this sense, the program is a collection of operators operating on a set of variables in a certain way. A given variable at a particular point during the execution of a program may be in one of three possible states:

U - Undefined
D - Defined
R - Referenced

A variable may be in the Undefined stage when due to language syntax and semantics the variable loses meaning or purpose. This may happen, for example, to the index of a DO loop outside the subroutine or block. A variable is defined when it is assigned a value. A variable is said to be referenced when it is fetched from storage. During the execution of a program., a variable is said to be in the hazard state H if it passes through any one of the following three sequences:

U - R
D - U
D - D.

The hazard state does not necessarily imply an error. It is a flag for checking for a possible error.

A variable X or a constant which restricts or limits another variable Y is called a modifier. The notation (X, Y) will be used to indicate that X modifies Y. In the following statement,

$$A(1,N) = B(M+L)*3 + R(S(K))$$

the following modification relationships hold

(1,A)	(M,B)	(K,S)
(N,A)	(L,B)	(S,R)
(B,A)		
(3,A)		
(R,A)		

This can be illustrated via a modification graph, as follows:

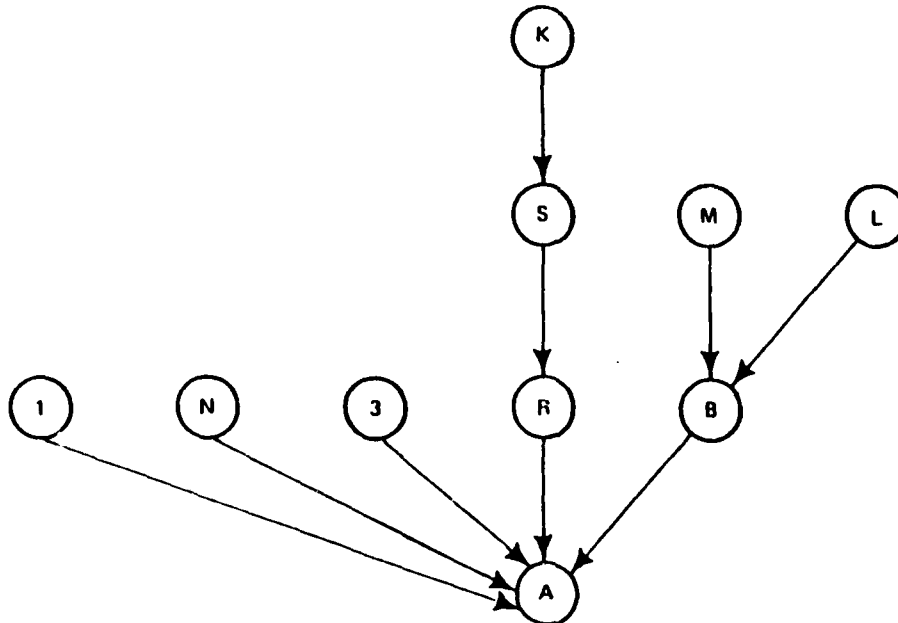


Fig. 2-11

Methodology

In data flow analysis attention is directed at the sequential pattern of definitions, references and undefinitions of values for variables. The actual values assigned or referenced are ignored, only the fact that definition, reference or undefinition was made is used. Two rules concerning the sequence of these events along each path from the start of a program to a stop are expected to be obeyed for each variable:

1. A reference must be preceded by a definition, without an intervening undefinition.
2. A definition must be followed by a reference, before another definition or undefinition.

Violation of the first rule called a type 1 anomaly should cause an erroneous result during program execution; moreover, in the case of FORTRAN it is a violation of the ANSI Standard. Violation of the second rule should result in a waste of time, but not an erroneous result.

Many things can cause a violation of either or both rules. Forgetting to initialize a variable is the most obvious cause of a

violation of the first rule. However, spelling errors, confusion of names, misplaced statements and faulty subprogram references also cause violations of this rule. The second rule may be violated when a programmer forgets that a variable is already defined or that it will not be used later. Many optimizing compilers remove this 'dead' variable assignment, assuming these to be the only causes. However, many common errors also cause violations of the second rule.

Data flow analysis of a program is of two types: dynamic and static.

Dynamic Analysis

In dynamic data flow analysis probes are inserted in a program to determine the points in a program at which a variable passes from one state (U,D,R) to another or is modified. The program is then executed and a record is made of each variable as it passes from one state to another. The hazard sequences are flagged and a modification matrix is constructed. An example of a modification matrix for an execution of a program is given below:

Modification Matrix											
		Variables									
		A	I	J	II	J1	TOP	STK1	STK2	TEMP	FLAG
CONSTANTS	0										X
	1		X	X			X	X	X		X
	2										
	A	X								X	
	I	X	X					X	X		
PARAMETERS	J	X		X							
	N								X		
	II		X					X			
	J1			X					X		
	TOP						X	X	X		
	STK1				X						
	STK2					X					
	TEMP	X									
	FLAG										X

Fig. 2-12

The columns represent variables and the rows contain constants and parameters which modify these variables. If a constant shows up at the top of this matrix as a variable which is modified, the program is in error. The modification matrix can be automatically scanned for any such errors.

The value of dynamic flow analysis in testing has only recently been realized. It is grossly under-utilized during testing. Although there is much current research in this area (23,29,30,44,58), for a practical global theory as it applies to issues of program testing and corresponding algorithms and tools has not been met yet.

Static Analysis

Static data flow analysis, on the other hand, is widely used during program testing for allegation checking. A static analysis avoids executing the program. It classifies all local and global variables and performs an exhaustive search for data flow anomalies. This type of analysis is usually done with the help of automatic tools.

DAVE, validation error detection and documentation system for FORTRAN programs, is a static analyzer developed by Leon J. Osterweil and Lloyd D. Fosdick. They discuss (50) how data flow graph information is used by static analysis to prove relatively strong allegations about FORTRAN programs. This research tool, typical of the fourth-generation automated tools, as Ed Miller calls them, is a system capable of detecting the symptoms of a wide variety of errors in a program. In addition, DAVE exposes and documents subtle data relations and flow within programs.

The messages issued by DAVE are divided into three categories: error, warning and general information. An error message is issued whenever DAVE is certain that a type 1 anomaly is present on an execution path. A warning message is issued whenever a type 1 anomaly might be present on an execution path. A warning message is issued if a type 2 anomaly is detected.

DAVE begins by dividing the program into program units. These are then divided into statements and statement type determination is made. Next, the subject program is passed to a lexical analysis routine which creates a token list to represent each of the program's source statements.

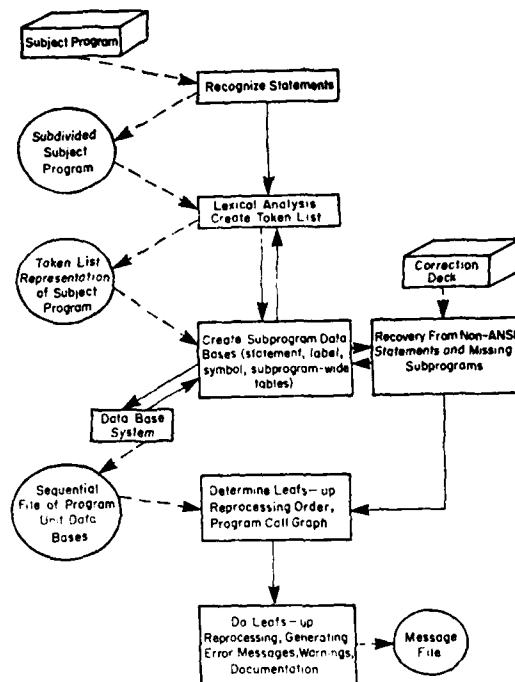


Fig. 2-13

As the token lists are created, comprehensive data bases for each of the program units are also created. Each of these data bases contains a symbol table, label table, statement table and a table of subprogramwide data. The symbol and label tables contain the same kind of information found in most compiler symbol and label tables, listing symbol and label attributes as well as the locations of all references to the symbols and labels.

During this lexical scan phase DAVE determines the input/output classifications of all variables used in each statement, except variables used as actual arguments in subprogram invocations, and loads this information into the statement table. The table of subprogram wide data for a program unit contains an external reference list containing all subprograms referenced by the program unit, as well as representations of all non-local variable lists; i.e., the program unit's dummy argument list and COMMON block lists. These lists are used to establish the input/output classification within the invoking program unit of all variables used as actual

arguments in invocations of this program unit. The external reference lists are used to construct the program call graph, a structure that indicates which subprograms invoke which other subprograms.

After all the program units of the subject program have been processed in this way, DAVE enters the main phase of documentation, analysis, validation and error detection. The program call graph is examined, and a leaf subprogram is selected for processing. Because this subprogram is a leaf, the input/output search procedures described above are immediately usable.

The local variables of the subprogram are analyzed first. An error message is generated for all local variables which are found to be strict input for the subprogram. The input/output classifications of the non-local variables of the subprogram are then determined. These classifications are printed, and also stored in the subprogram-wide table of the subprogram under study. Warning messages are also printed for all dummy arguments which are found to be non-input and non-output.

The system makes a special check of the usage of all DO-loop index variables following satisfaction of their DO's. If the first use of a DO index following DO satisfaction is input or strict input, an anomaly is indicated and a warning or error message is produced. These situations are detected by initiating an input category determination trace for the DO index where the trace is begun with the flow graph edge which represents the DO satisfaction branch.

The analysis of a non-leaf program unit is more complicated. Such a program unit will, of course, not be analyzed until all subprograms which it calls have been analyzed. Then DAVE can fill in all entries which had been left blank during the creation of the calling unit's statement table. Certain FORTRAN errors are detected as this proceeds. For example, mismatches between either the types or number of actual arguments in an invocation and the members of the corresponding dummy argument list are detected here. The use of an expression or function name as an argument to a subprogram whose corresponding dummy argument is either an output or strict output variable is also detected here.

DAVE also exposes concealed data flows through subprogram invocations. Concealed data flows result from the use of COMMON variables as inputs (or outputs) to (from) an invoked subprogram. Such situations are easily exposed by examination of the COMMON block variable lists in the subprogram table of the invoked subprogram.

If a COMMON block, B, is declared by a high level program unit which invokes a subprogram, S, in which the block is not declared, then the ANSI Standard specifies that B must still be regarded as implicitly defined in S provided that some subprogram directly or indirectly invoked by S does declare B. Hence data referenced by the variables in B may flow freely through routines which do not even make reference to B. Such data flows are noted and monitored by DAVE. In addition, DAVE is capable of printing the names and descriptions of all COMMON blocks whose declarations are implicit in a given subprogram. The algorithm for determining which blocks are implicitly defined in which routines involves a preliminary leafs-up pass through the program call graph and then a final root-to-leafs pass.

After all of the above described checking and insertion of input/output data into the statement table has been done, DAVE proceeds with the analysis of the variables, explicit and implicit, local and non-local, as described in the case of a leaf subprogram. The algorithm used here are generalizations of those described in the previous section.

Subprograms are processed in this way until the main program is reached. Processing of non-COMMON variables in the main program is the same as the processing of such variables in any non-leaf, but COMMON variables must be treated differently. Any COMMON variable which has an input or strict input classification for the main program must be initialized in a BLOCK DATA subprogram. If not, a warning message (if the classification is input) or an error message (if the classification is strict input) is issued. Similarly, if the last usage of a COMMON variable was as an output from a main program a warning message is issued.

CONFIDENCE LEVEL OF A TEST

Having completed a rigorous path testing program, there is still the more basic issue of what confidence do we have in the performance of the program. The test effectiveness measures give the coverage achieved by the test program for a given level k of DD paths in the program. The data flow analysis also sheds some light on the use of variables in the program. Based on the number of errors found in the execution of these paths, we will now determine a confidence level for the correctness of the program.

Suppose that P is a program which is meant to compute a function F with domain D. A testing strategy for P is a procedure for choosing a finite subset T of D. T is said to be reliable if:

$$P(x) = F(x) \text{ for all } x \in T \Rightarrow P(x) = F(x) \text{ for all } x \in D.$$

That is, T is reliable for P if T reveals that P is incorrect whenever P contains an error. Assume that there exists a hypothetical correct program P*. The differences between P and P* define the errors in P. William E. Howden has shown that path analysis testing is a reliable method for testing such programs, i.e., path testing reveals an error when one exists in the program (28). A patchword is defined as any of a number of machine instructions required to fix an error in the program P. Define

The degree of incorrectness of P

$$= \frac{\text{the number of patchwords required to correct the program P}}{\text{the number of machine instruction words in the program}}.$$

This is, of course, a hypothetical definition because the correct program P* is hypothetical. Based on the results of the path testing program, we can make a statistical inference about the degree of incorrectness of P.

A given DD path of length k in a program may be contained in several complete paths through the program. Equivalently, a set D of input data can be chosen such that each element d of D executes a different complete path through the program which contains the DD path of length k. Note that some of these complete paths may contain an error while others may not.

Suppose that a logic flow path testing program T consists of testing n_1 paths of length 1, n_2 paths of length 2, ..., n_s paths of length s. Suppose further that none of these paths is contained in another. Then the path testing program T chooses a sample of n complete paths through the program, where

$$n = n_1 + n_2 + \dots + n_s.$$

The problem of a complete path containing more than one DD paths of T, can easily be handled by requiring that a different complete path be chosen for each DD path contained in T. This corresponds to sampling without replacement.

Assume, now, that errors occur independently and are randomly distributed along different paths of the program, and that the number of patchwords required to fix an error is a random variable. Define, for each path i,

the number of patchwords required to correct the errors in P along path i.

$X_i =$

 the number of machine instruction words in the program
 $i=1, \dots, N.$

X_i are then independent and identically distributed random variables. Note that the number of machine instruction words in the program is a constant fixed for the program.

We will proceed to test the hypothesis that p, the degree of incorrectness of P, is less than or equal to a specified acceptable value p_0 .

Let x_i denote the observed value of X_i when path i is executed. Then, x_i is the total number of patchwords required to correct the program P divided by the total number of machine instruction words in the program. Hence, the observed degree of incorrectness of P is

$$S = \sum_{i=1}^n x_i.$$

The confidence level attained by the test program can then be obtained by using the Central Limit Theorem. We will reject the hypothesis $p = p_0$ at a given confidence level $100(1-\alpha)\%$ if

$$\sqrt{\frac{S - p_0}{\frac{p_0(1-p_0)}{N}}} > z_\alpha,$$

where z_α is the $100(1-\alpha)$ th percentile of the Gaussian distribution with mean 0 and variance 1.

Reinterpreting the same statistical analysis, having observed a value of the degree of incorrectness of the program P, we can have a confidence level of $100(1-\alpha)\%$ in the result where the value of $(S-p_0)/\sqrt{\frac{p_0(1-p_0)}{N}}$ is $100(1-\alpha)$ th percentile of the Gaussian

distribution with mean 0 and variance 1.

The above analysis may be carried out separately for the data flow paths chosen by the test program P.

Example

Suppose that a test program executes 10,000 logic flow and data flow paths through the program P. Suppose that $p_0 < .003$ is the acceptable degree of incorrectness, i.e., the program P is acceptable if it can be corrected with less than or equal to .003 x

(the number of machine instruction words in the program) patchwords. When the test program is executed, assume that the observed degree of incorrectness is

$$S = .004$$

Then,

$$\frac{S - p_o}{\sqrt{\frac{p_o(1-p_o)}{n}}} = \frac{.004 - .003}{\sqrt{\frac{.003(.997)}{10,000}}} = 1.83150.$$

This corresponds to 96.64% confidence level (24) as shown below.

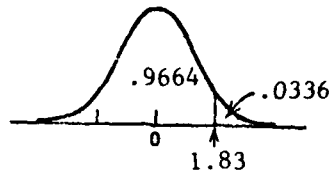


Fig. 2-14

So if the observed degree of incorrectness of the program based on test results is .004, we can still accept the program with about 97% confidence.

Specifications for a Test Program

The specifications of a test program for testing a computer program P can now be formulated in terms of

- p_o = the acceptable degree of incorrectness of P
- e = the acceptable error in estimating p_o
- $100(1-\alpha)$ = the desired confidence level.

The number n of paths through the program which must be tested in order to meet this specification is given by (24)

$$n = \left[\frac{z_{\alpha} p_o(1-p_o)}{e} \right]^2.$$

A typical specification for a test program would then read as:

The degree of incorrectness of P should be less than .003 with 98% confidence that the estimation error does not exceed .001.

This specification will be met if

- i) $n = \left[\frac{2.06}{.001} \times \sqrt{.003(.997)} \right]^2 = 12,692$ paths are executed through the program,
- ii) the number of patches required to fix the errors observed by executing these paths is less than .004 x the number of machine instruction words in the program.

If the program resources do not allow testing of so many paths, then the confidence level should be dropped.

ESTIMATING TIME TO COMPLETION

Fred Brooks in 1975 observed (5),

"In examining conventionally scheduled projects, I have found that a few allowed one-half of the projected schedule for testing, but that most did indeed spend half of the entire actual schedule for that purpose. Many of these were on schedule until and except in system testing."

Unlike the "black-box" methods of conventional testing, the more organized and disciplined method of path testing provides more control and visibility to the collected information relating to the quality of the software being tested and the resources being expended in collecting this information. In real and large systems, there is a point of diminishing returns on investment of program debugging efforts. Getting at absolutely all the errors in a real program is now recognized as a task requiring almost infinite resources. In 1972, in the 20th major version of their approximately 3 million dollar 360 Operating System, IBM officially reported approximately 12,000 new bugs in the system. At least 1,000 bugs had been discovered in each of the 20 releases in spite of 24 hour usage of the program for several years by thousands of installations (20). By tracking the history of bugs discovered in a program, we will now attempt to predict the time to completion for this program using the method of estimating the number of fish in a pond.

Estimating the Number of Fish in a Pond

In order to estimate the total number of fish in a pond, a reasonably large sample n of fish are caught and marked. These are then allowed to mix homogeneously with the population in the pond. Another sample of n fish is then caught. If m of this new sample

have markings on them, the total population in the pond can be estimated as

$$n / \left(\frac{m}{n} \right) \quad \text{or} \quad \frac{n^2}{m} .$$

Estimating Number of Errors Not yet Detected

The number of yet undetected errors in a program can be estimated by debugging the program with n artificially inserted bugs at the initiation of a debugging program. Now at any point during the program if only m of the artificial bugs have been detected while N of the real bugs are detected, the number of yet undetected errors can be estimated to be

$$\frac{n}{m} \times N \quad \text{or} \quad \frac{nN}{m} .$$

For example, 100 errors were inserted artificially and only 60 of these have been found so far while 300 real errors have been found, then the number of yet undetected errors of the same type can be estimated to be 500. Of course, the number of actual bugs in the program could really exceed this number if some types of bugs were not inserted in the program. For a more detailed discussion on how to make the artificially inserted bugs more representative, see Ref. (34, p. 36-39).

Errors Detected Vs. Time

If artificial errors are inserted in a program, the debugging curve plots the percentage of those detected as a function of time. Three experimentally achieved debugging curves are shown below (34).

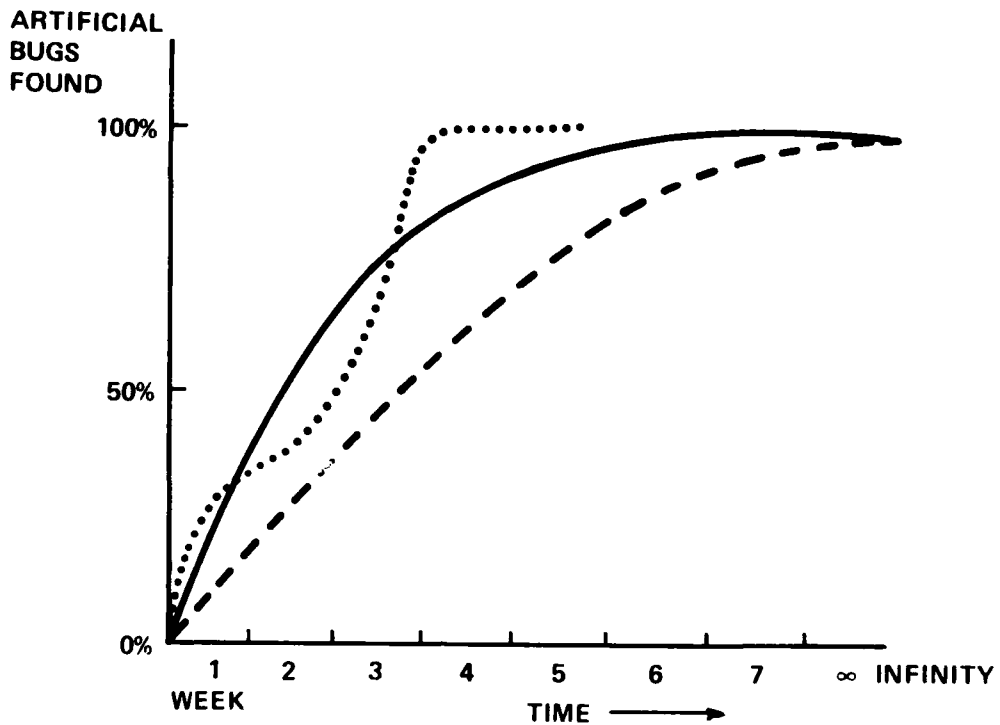


Fig. 2-15

Time to Completion

Using the debugging curve, the time to meet a certain quality specification of the program can be estimated. Regression analysis can be used to predict the time to achieve a certain percentage of correctness. Also just a visual inspection of the slope of the curve and the amount of artificially inserted errors caught so far can yield an estimate of the time required to catch a specified percentage of these errors.

The use of the metrics developed in this chapter can be organized into a complete software testing methodology. This is done in the next chapter.

Section 3

A SOFTWARE TESTING METHODOLOGY AS AN INTEGRAL PART OF SOFTWARE ENGINEERING

In the traditional software model, testing is a series of discrete steps associated with different phases. A programming project is viewed as a sequence of distinct phases: Definition, Design, Implementation, Testing, and Operation. Each project phase produces documentation which records the results of that phase and also disciplines the subsequent phases. The definition phase produces a specification and a statement of work before the design phase is initiated. The transition from the design to the implementation phase is demarked by Preliminary Design Review and Critical Design Review. PDR assesses the logical and technical feasibility of the design and CDR assesses the implementation and performance feasibility. The implementation phase comprises of coding, unit testing, and integration. Unit testing and development of system plans and procedures occurs during the implementation phase, while system testing itself constitutes the testing phase. The traditional approach has been strengthened by the addition of formal validation points at the end of each phase. This explicitly incorporates feedback loops into the development process.

The appeal of the traditional phased project model is simply that it is the contractual model. Software acquisition contracts and project management policy presently key their deliverables and milestones to the traditional model. The DoD now rigorously applies configuration management with successively approved baselines and formal change control procedures to the development of software systems. The model thus has a strong managerial justification. But in its technical formulation, it does not portray the evolutionary development of system releases. Such successive versions often accelerate user interfacing or accommodate a design to cost development strategy. Also the traditional model largely ignores the global feedback between software activities. For example, unit testing can expose coding errors, integration testing usually uncovers interface design errors and acceptance testing can reveal system deficiencies in function or in performance with respect to intended requirements.

McHenry in 1977 (37) proposed an alternative to the traditional management model by defining the concept of a continuum of specifying software activities where the full system is obtained via incremental construction and demonstration. This approach of "code-a-little, test-a-little" subsumes all testing activity. Using

the resources of the project and support facilities, the test procedures are themselves developed as a system, top down. This approach allows co-habitation with the traditional contractual model in terms of required reviews and deliverable documents.

Structured programming, top down design and implementation and path testing in the continuum model combine to form a complete software engineering model for software development and testing. In this section, we will examine how path analysis interacts with the other software engineering techniques and the effects of using these techniques on software testing.

STRUCTURED PROGRAMMING

Analogous to the proven sufficiency of using Boolean Algebra, of AND, OR and NOT gates for reading any logic circuit is the result that statement sequencing, IF-THEN-ELSE conditional selection, and DO-WHILE conditional iteration suffice as a set of control structures for expressing any sequential program logic. Dijkstra in 1972 introduced and developed structured programming as a "constructive" approach to "the process of program generation such as to produce apriori correct programs" (17). The well-structured program is more easily read, thus facilitating maintenance, modification, and correction. *Structured programming is of* invaluable help in testing the control structure of a program for the rules of program composition are limited to those that are well understood. The practice of structured programming, with its purely theoretical origins, "has been a catalyst for the review and change of software production practices." Structured programming, in less than a decade since its introduction has become a programming standard for all leading software developments.

TOP-DOWN IMPLEMENTATION

Top-down implementation is a hierarchical development of executable versions that model the final system. To obtain an executable system, program stubs are used. A program stub is some short code that permits any referencing code to continue execution. Thus a stub must meet any interface requirements. Stubs are later fully coded and, in turn, may reference other stubs. The simplest kinds of stubs are those represented by non-functional dummy code for debugging and testing purposes. Function stubs provide data to higher level segments through fixed parameters, simulation or some simplified skeletal procedures.

This approach was first advocated by Zurcher and Randell (64). H. Mills (38) refined it in the mid-seventies and was the principal influence in making it a production practice. Most extensively reported uses of this approach were on the New York Times Project, ASTP (Apollo-Soyuz Test Project), NASCOR (FAA National Air Space System Support Software) and the Skylab Ground Support Simulation (36).

TOP-DOWN IMPLEMENTATION IN THE CONTINUUM MODEL

In the continuum model, the top-down implementation becomes "code-a-little, test-a-little". Top-down implementation here requires interim executable versions of the program to be produced. Hence, some parts of the program are completely implemented while others remain as stubs. Implementing and integrating an interim executable version requires referencing some yet uncoded stubs. These stubs are later fully coded in the next executable version of the program and in turn may reference other stubs. Integration, therefore, becomes a continuous activity throughout development. Many errors that initiate rework are found during integration testing. By distributing integration activity over the entire development effort, the model serves as an executable check on the adequacy of the design of the system. Projects that defer integration testing are vulnerable to greater cost and schedule over-runs since the cost of error rework is 10 to 100 times higher and is accomplished in large part during the end of the schedule (43).

Yourdon and Constantine attribute the proven advantages of top-down implementation in the continuum model solely to its incremental testing aspect. William F. Ross in 1974 remarked,

"Code-a-little, test-a-little" seems to produce the majority of the increase in programmer productivity. This opinion is based on experience with three recent large scale real-time system development efforts at Hughes (54).

A COMPREHENSIVE SOFTWARE TEST METHODOLOGY

A comprehensive test methodology for a program which is perfectly structured with top-down design and implementation, can now be developed. This methodology can be applied to programs at any time during their evolution in the continuum model. In particular, it can be applied to interim executable versions of the program. One of Dijkstra's original arguments for top down design was that it resulted in modules that are simple enough to make it possible to test all logical paths (up to loop iterations) through each module (16).

The sequence of paths which can be traversed during an execution of the program are, of course, specifically defined by the control structure of the program from which they arose. A collection of digraph reduction algorithms (65) using the DD paths of the program define the level i paths for the program as follows:

A level 0 path leads from the entrance of the program to the output without employing any DD path more than once. In effect, the level 0 paths correspond to the "fall through" conditions extant in the program spine.

A level i path, $i > 0$, leads from an alternative predicate outcome along some level $(i-1)$ path, through a set of DD paths not present on any lower level path at a point earlier than the original (i.e., departing) DD path. A level i path, $i > 0$, represents iteration "over" a level $(i-1)$ path.

Typical computer programs contain only a few levels of iteration; the algorithms identify all of these automatically, and, in addition, identify the unique predecessors for each level i path. The collection of such level i paths forms a "tree" because of the precise ancestry relationships present. This tree is used as the basis for efficient organization of the search for meaningful test cases for a program.

Once the level i paths have been identified, the tree indicating their structure can be drawn. The tree is rooted in the program graph, and consists of a number of branches. The most interesting branches are the ones which have no successors, i.e., the terminal-branch level i paths. The following result has been established by M. Paige:

If a set of test cases traverses the DD paths which exist on the set of terminal-branch level i paths for a program at least once, then the set of test cases exercises every DD path at least once.

The implication of this theorem is as follows: because terminal-branch level i paths represent the "deepest buried" iteration structure of the computer program, testing the statements that lie at those locations assures the testing of the remainder of the program. For example, consider the triple iteration:

```
DO 10 I = 1, I1
  DO 10 J = 1, J1
    DO 10 K = 1, K1
      <action-statement>
10 CONTINUE
```

In this program fragment, the <action-statement> lies on a level-2 path residing on a terminal branch of the corresponding level i path tree. Testing this <action-statement> at least once also tests the remainder of the program fragment.

Path Analysis Test Approach

This approach to path analysis includes an analysis of the data flow paths and the logic flow paths of a program. Automated tools are used for the data flow analysis. The program is decomposed into a hierarchy of functional modules. All executable logic flow paths through a functional module which require less than or equal to k iterations of loops are tested at least once. Usually k is taken to be 2. This of course leaves some parts of a module untested because of complicated loop indexing operations and dependencies between loop bounds. But these modules can easily be identified and tested separately.

In the following paragraphs, we will see how the control structure of a program can be factored using a tree representation so that certain modules of a program can be tested together. This leads to method of testing the logic flow of the program such that test resources can be optimally allocated to different modules.

Hierarchical Decomposition of The Tree of a Program

The tree of the entire program with all its segments, CPCIs (Computer Program Configuration Items), CPCs (Computer Program Components) and modules provides a uniform structure-based representation of the total program text. The tree representation of the control space of a large top-down structured program follows the hierarchical structure of the program. If a segment P_1 of a program P is invoked by a statement sequence S, then the subtree emanating from the branch corresponding to S is a tree in its own right and corresponds to the control space of P_1 .

Program Factoring

When only certain modules of a program are to be tested together, the subtree generated by the trees corresponding to these modules can be used to do path testing. This method corresponds to creating a special subroutine that carries all of the program text contained in the set of modules which are to be tested. This method is called program factoring since the effect is to break a program into small enough pieces which can be tested thoroughly.

Weights for each branch of the tree can be defined as the number of statements in the original program that it represents. A

more appropriate method is to use the weights which represent the computational complexity of the sequence of statements represented by the branch. Such weights are developed in (59) and can be related more easily to the errors in the program.

Following are guidelines for selecting a subset of the collection of all feasible trees for comprehensive testing of a program (40).

- (1) A minimum weight for each subtree selected must be commensurate with the desire to accomplish treatment of the program in easily manageable portions.
- (2) Each leaf of the original tree must be included at least once in the set of subtrees selected.

The second criterion simply assures that the testing done accomplishes the coverage criterion already suggested as the minimum one, i.e., covering all branches. The first criterion is intended to equalize the difficulty of each individual test case (or test case class) considered. Once the feasible subtrees are found and weights are assigned, the problem of devising a good test structure reduces to finding a balanced covering subset tree.

Optimal Allocation of Resources

The hierarchical tree structure of a program can be more fully exploited now for optimal allocation of resources.

Define the weight of a subtree T_i as the sum of the weights of all its branches. Let the weight of subtree T_i be denoted by W_i .

A set of feasible subtrees from the tree structure can be identified, by program factoring, which represent manageable portions of the program that should be tested together for functional dependency. This set of subtrees should be such that each leaf of the original tree is included in at least one of the subtrees.

Define a subcollection $\{T_i; i=1, \dots, n\}$ to be a cover for the tree if each leaf of the original tree is included in at least one of the subtrees in the subcollection. Define it to be a minimal cover if no proper subset of $\{T_i; i=1, \dots, n\}$ is a cover.

A minimal $\{T_i; i=1, \dots, n\}$ for the tree can now be chosen such that

$$\prod_{i=1}^n w_i$$

is minimized over all the possible minimal covers. Call this an optimal cover.

An optimal cover $\{T_i; i=1, \dots, n\}$ for a program tree is not necessarily unique. Define the Optimum Cover $\{T_i; i=1, \dots, n\}$ for the tree to be an optimal cover which minimizes

$$\sum_{i=1}^n |w_i - \bar{w}|.$$

Such a cover would distribute weights most evenly over its component subtrees.

After the Optimum Cover $\{T_i; i=1, \dots, n\}$ has been chosen for a program tree, optimal allocation of resources to each of its component subtrees can be made proportionate to their individual weights.

Path Testing in the Continuum Model

The hierarchical framework presented above supports the logical relationships between the computer programs as they evolve in a continuum model. A stub in an interim executable version of the program is simply a leaf program in the program tree emanating from the branch corresponding to the invoking statement sequence S. When the stub is fully coded in the next version, it simply grows into a subtree in its own right emanating from the branch corresponding to S. Thus, a complete and effective test program can be carried out continuously throughout software development whenever an executable version of the program is available. The hierarchical framework also supports the mathematical formulation and validation of test effectiveness metrics in a natural manner. The DD paths of one executable version are simply subpaths of a more developed version.

Consider the program P and its tree given in Fig. 2-5 of Section 2. Once the tree of a program has been found, one can use it to assist in constructing reasonable test paths (39). The set of all subtrees that includes at least one leaf corresponds roughly to the set of all possible program flows. The first objective of analyzing the tree is to identify the set of structurally feasible flows. These are the ones that remain after the structurally infeasible flows are excluded. For example, in the tree shown in this example, it is not possible to have a flow which involves a sequence A and any other sequence; thus, any potential subtree that does not involve A alone is automatically structurally infeasible.

Of the trees that remain, some may be semantically infeasible, which means that a certain program action taken at one point results in a set of conditions that makes some other program action impossible. Although the obvious approach would be to examine sequence/predicate pairs in some natural order, it turns out that this is not really necessary. Because of the way the tree is constructed, only certain kinds of relations need be examined in detail. For purposes of illustration, we assume every path that is structurally feasible is semantically feasible.

For the program in this example, there are nine feasible subtrees; these are enumerated in Table 3-1. The subtree is indicated simply by noting the program sequences that belong to it; the column just to the right gives the weights associated with that subtree. The four possible covers are indicated by Xs in the last four columns. The notation D^k is used to indicate that the D segment is actually included a variable (but finite) number of times since D resides inside an iteration construct.

Tree No.	Program Segments	Weight	Cover No.			
	Present		1	2	3	4
1	A	4	X	X	X	X
2	B, E	10	X			
3	B, D^k , E	12		X		
4	B, F	5			X	
5	B, D^k , F	7				X
6	C, E	8				X
7	C, D^k , E	10			X	
8	C, F	3		X		
9	C, D^k , F	5	X			

Fig. 3-1

A very simple mechanism can be used to choose among the covers: simply multiply the weights for each element in the cover together and choose the product with the highest value. Other things being equal, a candidate cover set that distributes the weights as evenly as possible among the elements will tend to be chosen. Note that for this particular program each cover must involve the A segment since it is the sole member of an essential subtree.

The computations suggested above result in the following totals:

Sub-Tree No.	Cover No.	Weights	Product of Weights
1,2,9	1	4,10, 5	200
1,3,8	2	4,12, 3	144
1,4,7	3	4, 5,10	200
1,5,6	4	4, 7, 8	224.

A good starting point evident from this enumeration is the set of subtrees (1, 5, 6) since it represents a cover and has the best distribution of program weight.

Naturally, this example is an oversimplification, but the points to be made are clear. Algorithms for doing all of the computations described already exist, and while choosing an optimum cover may be something of a stumbling block, there are certainly plenty of algorithms around to serve as good initial choices.

Section 4

AUTOMATED TOOLS FOR PATH ANALYSIS

Automated tools make it possible to achieve a level of thoroughness in the testing process that would be almost impossible to accomplish manually. The basic principle of path testing is controlled execution of a program with known, predicted or observed, inputs and outputs, combined with internal measurement of the behavior of the program. The number of program discriminations required during testing a large computer program may easily approximate that required during the program implementation process. Naturally, current test programs fall short of that. The question is how to accomplish this process thoroughly and within the budget. Generally the program testing process involves the repetition of a few relatively simple procedures a large number of times. Hence, automated test tools become necessary.

Automated testing tools are program analyzers which typically operate on the source text of an entire software system and perform analyses in response to user's commands. The outputs are either reports giving the answer to user's questions or generate a specific system setup to assist the user in performing a particular testing action. Most of these tools like RXVP, FACES, PET, DAVE, JAVS, SQLAB, although claimed to be operational, are still in the development phase.

Automatic tools can be put in two broad categories: static and dynamic.

STATIC ANALYSIS TOOLS

Static analysis tools analyze a program without regard to its run time behavior and do not require the execution of the program. A static analyzer proves an allegation about a program. When the allegation is false, no instance of the feature that the allegation is protecting against have been found. Automation here reduces the cost and ensures that the proof of the allegation has been carried out comprehensively. Static analysis tools like FACES, RXVP, and DAVE apply checks to FORTRAN software systems and report allegation violations to the user. Two of the more widely used static analysis tools are described below:

Programming Standards Checker - applies a series of tests to determine if the program meets a pre-determined set of

"standards". The standards address issues like format, structure, organization, clarity and other related issues.

Static Allegation Analyzer - applies advanced techniques to analyze programs statically to prove important allegations about the program's behavior dynamically, for example, that "all variables are set before they are used".

DYNAMIC ANALYSIS TOOLS

Dynamic testing involves a series of steps that lead from the initial identification of test data and test objectives, through the actual execution process, to the final stages in which the outcomes of the test are analyzed and cataloged. There are several major categories of automated tools that help in the dynamic testing process. In addition, there are many tools of only secondary importance that can be quite useful also. Generally speaking, the role of the tools is secondary to that of the human tester, who ultimately must decide which steps should be taken next. Following are some categories of automated tools for path testing.

Automated Test Facility - constructs a stubbed and standard input/output environment for a single program or a set of programs.

Reliable Test Analyzer - checks out the reliability of a test case in protecting against assignment and/or control errors, thereby helping to maximize the surety attained in a rigorous testing activity.

Robust Language Processor - automatically rewrites a program so that it "can't fail" during execution without first telling the user exactly how it did fail. Used during debugging and checkout testing to (a) isolate mistakes and (b) minimize lost execution time.

Test Planning/Status - a tool for management for the test program. This tool provides continual status checks.

Automated Modification Analyzer - automatically analyzes a program set for the potential impact a proposed program change will have on the testing process, and advises what re-testing will have to be performed as a consequence of the change.

Dynamic Assertion Processor - the programmer puts assertions about the way programs are supposed to behave into the code (they're treated as comments by the compiler), but the tool makes them report on when the assertions are violated.

Self-Metering Instrumentation - the programs are completely instrumented (but logical integrity is preserved) so that all conceivable interesting data about each programs' execution-time behavior is collected and reported (under user command).

Testing Difficulty Estimator - tells, according to detailed program structure analysis, which programs in a set are the easiest and most difficult to test and helps management control the testing resource better.

Automated Test Data Generation - automatically generates test data that meets specific objectives, using advanced heuristic processes that have high success ratios for this very difficult problem.

Two other tools that are also important during dynamic testing activity are test data/file generator and output comparator. A test file generator is a stand alone package which constructs input files containing pseudo data for use by the program being tested. An output comparator tool is used to compare two successive versions of a program's execution output to identify the differences.

The most often used dynamic testing tool is the execution verifier. During the testing activity a test effectiveness standard would be defined. The execution verifier provides the tester specific information about the effect that a test has on the internal control flow behavior of the program. This is usually done with automatic instrumentation of the program text, an execution processor and a report generator. In operation, the system follows instructions stated by the user in analyzing and instrumenting source programs; the instrumented versions are compiled and loaded. During execution, the instrumentation software omits data that are collected and recorded by the run time package. After execution, the post-processing system analyzes the information collected by the run-time package and produces coverage reports. The reports give the test effectiveness measures and signal when a program component is not tested.

Several tools which cover the entire range of dynamic testing are currently in development. Some of these have already been in existence for several years and are still in their initial operational phases. JAVS, the Jovial Automatic Verification System, was developed in 1976 by General Research Corporation and is in its initial operational phases at Rome Air Development Center (RADC). Another tool, SQLAB - A Software Quality Assurance Laboratory, is a complete automated test tool for programs written in PASCAL, FORTRAN, VPASCAL and IFTRAN. There is currently a gap between exhaustive testing of all potential program paths and the

capabilities of these automated tools to handle these paths through a program. Advance research efforts are being conducted in several areas in an attempt to help bridge this gap (57, p. 212). Most current tools like the one described below--SQLAB, deal only with statement testing and branch testing. But these are of invaluable help in identifying paths of any length through a program.

SQLAB - A SOFTWARE QUALITY ASSURANCE LABORATORY

SQLAB is an automated tool, developed by GRC, for analyzing source programs. It is claimed to be useful during all phases of the software development cycle, but it is of special importance during formal test and check-out of computer programs. During this phase, input data is usually available to run the program to obtain certain outputs. What happens in between the input and the output is mostly invisible. SQLAB gives a visibility and an interactive ability to observe the flow of variables and logic at different branch points in each of the modules of the computer program. It is claimed to include a variety of new constructs and allow the addition of executable assertions to the program, yielding a more comprehensive static analysis and a much more useful dynamic analysis for execution testing and formal verification. Documentation related to SQLAB can be obtained from GRC. This documentation is scanty and, therefore, the tool is difficult to use if one is unfamiliar with it.

SQLAB is a software system which reads source code text as data, either from cards or from a card image file. The type of processing to be performed on the source code is specified through commands that are input interactively or on cards. During an initial run, a data base or library is constructed which contains information about each module submitted for analysis. SQLAB has several components (OPTIONS) which extract information from this library and produce reports.

The command which controls the type of processing to be done by SQLAB is

OPTION(S) = <list>

The possible options are as follows:

- 1) LIST - produces an enhanced source listing of each module, which shows the number of each statement, the levels of indentation, and the DD-paths. A report from this option is given below:

STATEMENT LISTING

SUBROUTINE BSORT (NUM, ARRAY)

NO.	LEVEL	LABEL	STATEMENT TEXT...	FLPATHS
1			SUBROUTINE BSORT (NUM, ARRAY)	(1)
2			INTEGER ARRAY (100)	
3			INTEGER SMALL	
4			INPUT (/ I / NUM)	
5			IF (NUM .GT. MAXNUM)	(2- 3)
6	(1)		• N = MAXNUM	
7	(1)		• CALL ERROR (NUM)	
8			ELSE	
9	(1)		• N = NUM	
10			ENDIF	
11			I = 2	
12			WHILE (I .LE. N)	(4- 5)
13	(1)		• IF (ARRAY (I - 1) .LE. ARRAY (I))	(6- 7)
14	(2)		• I = I + 1	
15	(1)		• ELSE	
16	(2)		• SMALL = ARRAY (I)	
17	(2)		• ARRAY (I) = ARRAY (I - 1)	
18	(2)		• J = I - 2	
19	(2)		• NEXIT = 0	
20	(2)		• WHILE (NEXIT .EQ. 0)	(8- 9)
21	(3)		• • IF (J .GE. 1)	(10- 11)
22	(4)		• • • IF (SMALL .LT. ARRAY (J))	(12- 13)
23	(5)		• • • • ARRAY (J + 1) = ARRAY (J)	
24	(5)		• • • • J = J - 1	
25	(4)		• • • • ELSE	
26	(5)		• • • • NEXIT = 2	
27	(4)		• • • • ENDIF	
28	(3)		• • • ELSE	
29	(4)		• • • NEXIT = 1	
30	(3)		• • • ENDIF	
31	(2)		• • ENDCWHILE	
32	(2)		• ARRAY (J + 1) = SMALL	
33	(2)		• I = I + 1	
34	(1)		• ENDIF	
35			ENDWHILE	
36			CALL PRNT (N, ARRAY)	
37			OUTPUT (/ I / NUM, (ARRAY (I), I = 1, NUM))	
38			RETURN	
39			IFLAG = .TRUE.	
40			END	

Fig 4-1. Statement Listing

- 2) **STATIC** - produces a Static Analysis Report on the consistency of variables and an Invocation Space Report for each module. This analysis is designed to uncover inconsistencies in the use of variables and in the structure of a program. When such an inconsistency is discovered, the analysis detects an error or indicates the possibility of an error. A more comprehensive analysis is possible when assertions have been added to the user's program.

The Static Analysis Report has two sections. The first section is a Statement Analysis and the second section a Symbol Analysis. The Statement Analysis reports on the following types of checking:

- Graph checking, which identifies possible errors in program control structure, such as unreachable code.

- Call checking, which validates actual invocations against formal declarations and notes inconsistencies in type and number of parameters as warnings or errors.
- Mode and type checking, which identifies possible misuse of constants and variables in expressions, assignments, and invocations; inconsistencies are designated as warnings or errors.

The second section is a Symbol Analysis which lists each symbol for which an inconsistency was found by name, scope, type, and mode and identifies its use (INPUT, OUTPUT, or BOTH). When a variable is set but never used, there is a warning; however, if a variable is used before it is set, this condition constitutes an error. The summary tabulates the total number of errors and warnings.

The Invocation Space Report shows all invocations, along with the statement numbers, to and from the specified module. It is useful in examining actual parameter usage.

- 3) INPUT/OUTPUT - checks in addition to Static analysis, the consistency of the use of variables as defined in INPUT and OUTPUT assertions against the actual use. The actual use is determined by the data-flow analysis, which is a path-by-path analysis of the use of variables.

Executable assertions make possible a dynamic checking of the value of variables, which may uncover errors not found with static consistency checks. This dynamic checking includes tabulation of the value as of global variables each time a module is invoked.

- 4) UNITS - checks, in addition to static analysis, the consistency of units that have been defined in a UNITS assertion.
- 5) LOOP - analyzes paths through loop structures, searching for particular deficiencies which may indicate an infinite loop construct and produces a Loop Analysis Report only when loop termination is not assured.

- 6) STUB - builds a stub library containing information about the actual and formal parameters of modules to aid the static analysis on stubs.
- 7) COVERAGE- analyzes test coverage of segments for all instrumented modules that are invoked. There are FOUR different OPTIONS in this category.
- 8) DATA - gives detailed data flow analysis including LIST, STATIC, I/O, UNITS, and LOOP
- 9) INSTRUMENT - Inserts probes into the source code and produces a DD-Paths definition report (Fig 4-2)

OPTION = INSTRUMENT

OPTION = INSTRUMENT

DD-PATH, LABEL, LITERS	SUBROUTINE EXAMPLE (INFO, LENGTH)	
1	SUBROUTINE EXAMPLE (INFO, LENGTH)	** LCPATH 1 IS PROCEDURE ENTRY
2	C	
3	C	
4	C	
5	IF (INFO .LE. 10 .AND. LENGTH .GT. 0)	** LCPATH 2 IS TRUE BRANCH ** LCPATH 3 IS FALSE BRANCH
6 (1)	CALL CALLER (INFO)	
7	ELSE	
8 (1)	LENGTH = 50	
9	ENDIF	
10	CASEDEF (INFO * 6)	** LCPATH 4 IS BRANCH OUTWAY 1 ** LCPATH 5 IS BRANCH OUTWAY 2 ** LCPATH 6 IS BRANCH OUTWAY 3
11	CASE (14)	
12 (1)	LENGTH = LENGTH - INFO	
13	CASE (17)	
14 (1)	WHILE (INFO .LT. 20)	** LCPATH 7 IS LOOP AGAIN ** LCPATH 8 IS LOOP ESCAPE
15 (2)	REPEAT	
16 (2)	INVOKE (COMPUTE LENGTH)	
17 (2)	IF (LENGTH .GE. 30)	** LCPATH 9 IS TRUE BRANCH ** LCPATH 10 IS FALSE BRANCH
18 (4)	INVOKE (PRINT-RESULTS)	
19 (2)	ENDIF	
20 (2)	UNTIL (LENGTH .LE. INFO)	** LCPATH 11 IS LOOP ESCAPE ** LCPATH 12 IS LOOP AGAIN
21 (2)	INFO = INFO + 1	
22 (1)	ENDDO	
23	CASEELSE	
24 (1)	WHILE (LENGTH .GT. 0)	** LCPATH 13 IS LOOP AGAIN ** LCPATH 14 IS LOOP ESCAPE
25 (2)	INVOKE (COMPUTE LENGTH)	
26 (1)	ENDDO	
27	INVCASE	
28	ELCER (PRINT-RESULTS)	** LCPATH 15 IS PROCEDURE ENTRY
29 (1)	PRINT 1, INFO, LENGTH	
30 (1)	FORMAT (10A, 15I, 20I, 15I)	
31	ENDBLOCK	
32	ELCER (COMPUTE LENGTH)	** LCPATH 16 IS PROCEDURE ENTRY
33 (1)	LENGTH = LENGTH - 10	
34	ENDBLOCK	
35	RETURN	
36	END	

Fig 4-2 DD-Path Definition Report

10) **SUMMARY** - produces a report summarizing test coverage for all instrumented modules. When multiple test cases are involved, the **SUMMARY** report shows data from the current test case and the immediately preceding test case. When the end of the trace data is encountered, a cumulative summary of all test cases is produced. The summary report lists the following information (see Figure 4-3)

- Test case number
- Module names and numbers of DD-paths of length one
- Number of module invocations, number of DD-paths of length one and percent coverage for this test case
- Cumulative number of invocations, number of DD-paths traversed of length one, and percent coverage for all test cases.

Coverage Analysis
OPTION = SUMMARY

OPTION = SUMMARY

SUMMARY - THIS TEST						CUMULATIVE SUMMARY			
TEST CASE	MODULE NAME	NUMBER OF DD-PATHS	NUMBER OF INVOCATIONS	DD-PATHS TRAVERSED	PER CENT COVERAGE	NUMBER OF TESTS	INVOCATIONS	TRAVERSED	COVERAGE
1	MAIN	5	1	3	60.00	1	1	3	60.00
	CLASS	90	1	26	28.89	1	1	26	28.89
	EXAMPL	16	1	6	37.50	1	1	6	37.50
	CALLER	3	1	2	66.67	1	1	2	66.67
	SUBALLS	122	1	37	30.33	1	1	37	30.33
2	MAIN	5	0	2	40.00	2	1	5	40.00
	CLASS	90	1	10	11.11	2	2	29	29.99
	EXAMPL	16	1	6	37.50	2	2	7	43.75
	CALLER	3	0	0	0.00	2	1	2	66.67
	SUBALLS	122	1	26	21.31	2	1	41	33.61
3	MAIN	5	0	2	40.00	3	1	5	40.00
	CLASS	90	1	34	37.78	3	3	42	42.86
	EXAMPL	16	1	6	37.50	3	3	15	43.75
	CALLER	3	0	0	0.00	3	1	2	66.67
	SUBALLS	122	1	47	38.52	3	1	63	51.64
4	MAIN	5	0	2	40.00	4	1	5	40.00
	CLASS	90	1	34	37.78	4	4	42	42.86
	EXAMPL	16	1	6	37.50	4	4	15	43.75
	CALLER	3	0	0	0.00	4	1	2	66.67
	SUBALLS	122	1	42	34.43	4	1	63	51.64

Fig 4-3. Multiple Test DD-Path Summary

11) DETAILED - produces a report which shows a breakdown of individual DD-path coverage. A single-testcase report, and a cumulative report which is generated after all the individual test case reports, provide the following information:

- Module names
- Test case numbers
- List of DD-path numbers, with an indication of those which were not executed, a graphical representation of the number of executions, and an itemized list of the number of executions
- Overall module coverage data

12) NOTHIT - lists DD-paths not executed for all the instrumented modules. A NOTHIT report is generated (Fig 4-4) which lists the following information:

- Module names
- Test case number
- Number of DD-paths not traversed for this test case and for all test cases
- DD-paths not traversed for this test case and for all test cases.

Coverage Analysis

OPTION = NOTHIT

OPTION = NOTHIT

MODULE 1 TEST 1 PATHS 1										LIST OF DEFINITION TO DECISION PATHS NOT EXECUTED									
NAME 1 NUPBLM 1 NOT HIT 1																			
KPAIR > 1 3 1 3 1 1 2 5																			
1 CUMUL 1 1 1 2																			
KCLASS > 1 3 1 64 1 3 5 4 7 10 13 17 17 20 21 26 28 30 32 34 36 37 38 39 40																			
41 42 44 46 48 50 51 52 53 54 57 58 59 60 62 63 66 67 68 70																			
72 73 74 75 76 77 78 79 81 84 85 86 87 88 89 90 91 92 93 94																			
95 96 97 98																			
1 CUMUL 1 50 1 3 5 4 7 10 13 17 19 20 21 26 30 32 34 36 37 38 39 40 42																			
44 45 51 52 53 54 56 57 62 63 66 67 68 73 74 75 76 77 78 79																			
81 84 86 87 88 89 90 91 92 93 94 95 96 97 98																			
KEXAMPL > 1 3 1 5 1 2 4 6 13 14																			
1 CUMUL 1 1 1 4																			
KCALLER > 1 3 1 3 1 1 2 4																			
1 CUMUL 1 1 1 2																			

Fig 4-4. DD-Paths not Executed

- 13) VGG(Verifiable Condition Generator) - The program verifier uses the program paths of a module in generating the verification conditions, so it is necessary to perform a structural analysis of the module before setting up the remaining VCG commands. This analysis adds to a data base library a description of the program's structure in terms of DD-Paths. The command to perform this preliminary analysis is:

OPTION = VCG.

Two Steps are required to perform Verification Condition Generation (VCG). The first is a preliminary step to obtain necessary data, and the second is the actual generation.

The command to generate a verification condition is

VCG,PATH = <number of paths>,<path list>

where the path list consists of a set of DD-path numbers. For example, the command to cover the path from program entry to the first decision statement would be

VCG,PATH = 1,1

To cover DD-paths, 2,4 in a loop construct, the command would be

VCGG,PATH = 2, 2, 4

Usually several path commands are given at once.

There are several other interactive commands that a user may choose from a COMMAND MENU consisting of

VCG, SIMPLIFY

VCG, REPLACE

VCG, EXPRESSION

VCG, RXVP.

Although these capabilities are claimed to be operational by GRC, users have experienced several difficulties with them. These are at the state-of-the-art and will be most useful for verification of programs whenever they become available.

SQLAB is currently installed on a CDC 7600 at the Advanced Technology Division of Ballistic Missile Division in Huntsville, AL. A copy of the program on tapes has been acquired by MITRE. If all the capabilities that SQLAB was designed for were operational, it would be the most useful tool in existence for debugging, analysis, testing and verification of programs in FORTRAN, IFTRAN, PASCAL and VPASCAL (Verifiable PASCAL). SQLAB has very limited analysis capabilities for PASCAL programs. It was designed to automatically translate PASCAL programs into VPASCAL for comprehensive analysis of data and logic. But the SQLAB preprocessor for this automatic translation was deleted by GRC before submitting SQLAB to BMD. So this translation must now be done manually. Programs written in PASCAL have another major disadvantage. PASCAL has not yet been standardized and certain conventions in punch and symbology are not uniform. The attempts to use SQLAB on the Automatic Speaker Verification algorithm of Base Installation Security System/Entry Control System have surfaced some of these fundamental problems.

Also SQLAB does not have the capability of analyzing program paths that have more than two decision nodes. This part of the analysis must be done manually, although information from the level one DD-path analysis would reduce the task significantly. Subsequent test effectiveness measures can then be developed to measure test coverage at different levels of the hierarchy.

Section 5

APPLICATION TO AIR FORCE PROGRAMS

Software Quality Assurance has been an area of serious concern on DoD contracts. Computer resources, in particular computer programs, or software, are usually major elements of defense systems. Thus the quality of the defense system and its performance are highly dependent on the quality of the software program. In the Defense System Software FY79-83 Research and Development Technology Plan, software quality assurance has been given special importance. The plan has now been approved by the R & D Technology Panel on Software Technology which was established to assist the Director of Defense Research and Engineering in its execution. Through representation on the R & D Technology Panel, DoD has formulated the following specific objectives regarding software quality:

1. Development of quantitative and qualitative measures of software quality
2. Quantification of reliability
3. Development of methods and tools for testing to determine adherence to computer program requirements within stated tolerance limits
4. Establishment of a uniform error collection and analysis methodology
5. Development of tools and techniques for providing consistency of computer programs and their specifications.

DoD SOFTWARE QUALITY REQUIREMENTS

The mandatory DoD directives, regulations and instructions which establish policy and procedures for software quality and its management are shown in Fig. 5-11(11).

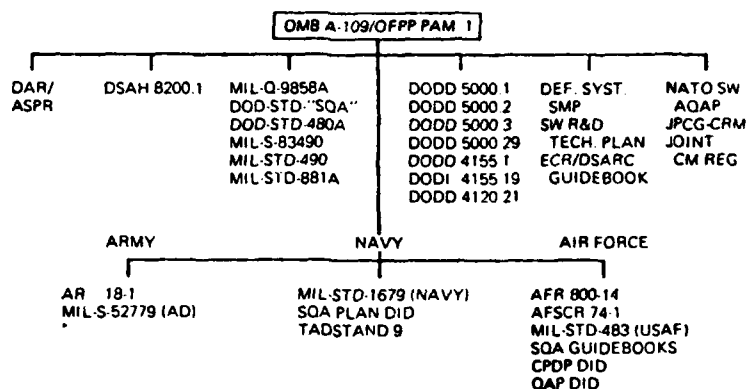


Fig 5-1

The major elements of this framework are:

DAR (ASPR)
DoDD 5000.1, 5000.2, 5000.3
DoDD 5000.29
DoDD 5010.19
DoDD 4155.1

The policies and procedures in these publications address the major system acquisition process, test and evaluation, computer resources, configuration management, and quality. The Defense Acquisition Regulation (DAR), formerly known as the Armed Services Procurement Regulation (ASPR) contains quality assurance policy and procedures. The DAR contains a quality program clause for insertion in contracts, and also states the procurement quality assurance responsibilities of the contractor and the Government, assigning responsibility for quality assurance performance to the contract administration office. DoDD 5000.29 addresses software acquisition exclusively, and includes policy specific to software reliability and correctness. The others explicitly include software within their scope. DoDD 4155.1 is devoted exclusively to quality assurance. A DoD level standard dealing exclusively with software quality assurance is still under preparation and is expected to be issued soon. DoDD 5000.3 deals exclusively with test and evaluation.

DoDD 5000.3

DoDD 5000.3, "Test and Evaluation," 11 April 1978, provides policy for test and evaluation of defense systems over their life cycle, and applies to software as well as hardware components. This is a recent revision and includes a section which amplifies key policy aspects as they apply specifically to software. DoDD 5000.3 is oriented primarily toward demonstrating quality, in particular

the performance aspect, for the evolving and resulting system. For test and evaluation involving software, the DoDD 5000.3 requires that:

1. Quantitative and demonstratable performance objectives shall be established for each software phase.
2. The decision to proceed to the next phase shall be based on quantitative demonstration of adequate software performance, using test and evaluation.
3. Prior to release for operational use, software shall undergo operational testing under realistic conditions sufficient to provide a valid estimate of system effectiveness and suitability in the operational environment.
4. Operational test and evaluation agencies shall participate in early stages of software planning and development to insure adequate consideration of the operational environment and operational objectives.

SOFTWARE QUALITY ASSURANCE AND SOFTWARE TESTING

In the Air Force contracting environment where the role of the Air Force is only to monitor the contractor and to interface with the user, software testing must bear the major burden of software quality assurance. In spite of the new disciplines, procedures, tools, and techniques, the software product, up until its final functional performance, remains somewhat of an intangible and abstract item. Project managers who can visit an engineering laboratory and view the evolving hardware and assess its progress, problems and risks, are unable to attain such insight into the workings of the software being developed. Barney M. Knight says "when it comes to software visibility, people have only one eye and hence no depth perception". Hence, software quality assurance must rely most heavily on data, mostly test data which is obtained by static or dynamic analysis of the program and which gives visibility into the structure of the program.

At present, in Air Force programs, software quality assurance is usually implemented by:

1. Contractual specification of both computer program requirements and requirements for a program to manage software quality.

2. Contractor monitoring and assessment of the effectiveness of his implementation of the quality program, usually through testing and reviews.
3. A software testing program which usually requires the program to execute for a specified amount of time to demonstrate the contractor's compliance with software quality requirements.

THE ROLE OF PATH TESTING METHODOLOGY IN AIR FORCE PROGRAMS

As part of the QA plan, the path testing methodology fulfills the first three objectives of the R & D Technology Plan by quantifying the specifications of an acceptable software test plan and the acceptance criteria. It provides a needed organization and discipline to the software test program which otherwise consists of "tricks" and techniques for "black box" testing. A uniform method of error collection is necessary for proper implementation of this software testing methodology. The proper automated tools, if available, can be of invaluable help. However, as discussed in the previous section, most of these are still in their initial operational testing and evaluation phase and as such cannot be imposed on a contractor. Software testing commences with contract award and continues until the computer program is accepted by the Air Force for operational use. Achievement of the software test objectives requires the establishment of checks and balances at each major step in the testing program, during the generation of test plans and procedures, during the conduct of tests and in the analysis and summary of test results.

The methodology is compatible with the established Air Force contracting environment and can be used during development or during system testing. Development testing is that testing which is performed by the contractor, at his discretion, to assist in the development of the software and to provide visibility regarding the progress of the development. Therefore, during development, software testing should be a continuous activity. If path testing is used, confidence levels in the correctness of any module of an interim executable version of the program can be determined, based on available or generated data, as discussed in Section 2 of this document. This will assist in bringing problem areas in the forefront, as they develop. A module should not be released until it reaches a certain confidence level compatible with the overall specified confidence level. The required confidence levels in the correctness of subprograms may be based on the criticality of the functions of that program, in such a way that an overall specified confidence level can be achieved for the entire program. Critical paths

through the program or high risk modules can be subjected to more exhaustive testing. Decisions like whether a low confidence module should be reprogrammed or fixed can be made. Another benefit is a more scientific method of predicting time to completion based on the history of errors uncovered in testing a particular module. More realistic statistical estimates of time to completion can be achieved using the estimation methods like "Number of Fish in a Pond" method of Section 2, thus resulting in better planning and management of resources.

The major benefits of path analysis are more evident in system testing. A complete system test methodology based on path analysis techniques can be developed as follows:

Effectiveness of Test Data. A great deal of real or simulated data are generated during program development. Before the generation of further test data, the effectiveness of this data set should be measured. By executing the program with this input data, TEMs of different levels can be computed. The portions of the program not yet covered can be identified.

An Upper Bound for The Amount of Testing. The first question that faces a test programmer is: how much testing is needed to meet specifications. Theoretical minimum for completely testing the program can be computed using the Holt-Paige measure. However, in practice the real constraints are the available time and money. Testing a minimum cover for all program paths in every module is almost never possible. But the theoretical limits can guide in allocating appropriate proportions of resources to different modules.

Critical Paths and Modules. The critical paths and modules in terms of functional performance can be separated at this time for more exhaustive path testing.

Specifications for The Test Program. The specifications of the test program should now be formulated in terms of the acceptable level of incorrectness, the acceptable error in estimation of this level, and the desired confidence level for this estimation. These specifications determine the amount of testing required in terms of the number of data flow and logic flow paths required for the test program in order to meet these specifications. A statistical test of hypothesis should be set up in order to determine the acceptance criteria (as explained in Section 2) in terms of patchwords required to fix the errors discovered by the test program.

An equivalent way of defining the specifications of the test program is by specifying the acceptable level of incorrectness

of the program, the required amount of testing in terms of the number of logic flow paths and data flow paths to be tested, and the acceptance criteria in terms of the number of patches required to fix the errors discovered during the test program. The confidence level of this test can then be determined as in Section 2.

For those modules of the program which have been identified as critical, separate specifications can be developed which will require more testing in terms of patchwords required to fix the program.

Optimal Allocation of Resources. After the resources required for the critical modules have been estimated, the tree structure of the program can be utilized for allocating the remaining resources optimally. This is described in Section 3 in more detail.

Path Testing. DD path analysis can now be performed. Data can be generated to execute paths at the chosen level. Portions of the program which have not been tested at the appropriate level can be identified and more data generated for their execution. TEMs can now be generated and the confidence level for the test determined. If the results are not acceptable, testing may continue until resources are exhausted or a decision can be made to reprogram portions of the program being tested. The observed confidence level and TEMs may be compared to the values set in the acceptance criteria.

The above test methodology presents a foundation for a complete, disciplined and quantifiable approach to software testing. It is very practical. It has been interfaced with the AF procurement program and written up in a Request For Proposal (RFP) RFP package for procurement of BISS/ECS. The lack of empirical data and established automated tools to generate test data limit its application at present. However, several automated tools have been developed which are in their initial operational phases and which will make it possible to use path analysis based testing on most AF programs.

Some of the basic problems which remain are brought out in the next section. These problems can, at present, be dealt with only heuristically at best. With the aid of automated tools discussed in Section 5, the potential of this methodology can be greatly enhanced.

Section 6

PROBLEMS AND TRENDS FOR THE 80s

The basic principles of software testing - repeatability, determinism, reliability, completeness and decomposability - are well established both in theoretical and practical terms. The foundations of a formal theory have been laid in the fundamental theorem of software testing stated and proven by Goodenough and Gerhart, and its extensions by Howden. The testing methodology presented here combines automated program analysis, reliable path testing and hierarchical decomposition of programs to offer a practical basis for program testing. However, many technical issues remain. There is also the practical problem of dealing with an immense number of paths which can be defined through any realistically sized program. The difficulty is that the number of tests that constitute the equivalent of program proof is either too large or impossible to construct economically. The complexity of the computations makes manual analysis virtually impossible, making the use of automated tools essential. There is an intense and growing interest in the programming community in these problems whose solution could turn software testing into an engineering discipline. The major problems and needs targeted for the 1980s are identified by Ed Miller (39). These are described in the following paragraphs under three categories: Theory, Methodology and Automated Tools.

THEORY

Program testing has been proven to be effectively the full equivalent of program proof of correctness provided the appropriate auxiliary analysis can be performed to select test data. The use of graph theory to model the control structure and data structure of a program for selection of appropriate test data has also been established. But the following problems still remain:

Test Data Generation. The test data generation problem, i.e., given any feasible DD path of a program to generate input data which exercises that path, is unsolvable (27). Much of the current work in test data generation involves systems which automate part of the path analysis testing strategy. In some of the systems the user selects program paths and the computer generates descriptions of data which cause the paths to be followed during execution.

Hierarchical Decomposition of a Program Tree. A uniform methodology for hierarchical decomposition of a multi-module program tree into clusters of subtrees for effective testing is needed which is independent of the programming language and the machine used. Systems that convert unstructured programs into structured programs practically use such a methodology. However, the methodology has to be refined to be language and machine independent and to yield a cluster of subtrees for effective testing.

Partitioning the Test Data. A general method of defining a subset of a given complete input data set which is itself a complete input data set for a particular cluster of modules of the program is needed.

Data Flow Analysis. The theory of data flow is underutilized for program testing. At present the only practical application of data flow graphs is in the area of allegation checking. Data flow graph algorithms and data structures need to be developed to detect the errors in the data flow.

Completeness. Given a data set, a method is needed for determining the minimum set of additional constraints that must be met by a complementary data set so that the test is a reliable proof against all types of software errors.

METHODOLOGY

The following developments are needed to establish a complete and practical methodology for formal testing:

Infeasible Paths. An infeasible path is one which cannot be executed by any set of input data. Paths may be structurally infeasible or semantically infeasible. A DD path is structurally infeasible due to the structure of the program, and semantically infeasible because a certain program action taken at one point results in a set of conditions that make some other program action impossible. Given a DD path, one needs to know whether it is infeasible. This is a tricky problem which needs to be resolved.

Error Analysis. The goal of software testing is to detect the errors in a computer program. Error data on large software programs has been collected all through the seventies. However, the nature of software errors is not yet well understood. Surprisingly, very little is known about the effectiveness of software testing in detecting different types of errors. A uniform error collection methodology must be developed so that software can be modelled as a potentially erroneous process which can be used to predict failure probability from level of testing effort.

Established Level for Minimum Test Effectiveness. Uniform and consistent Government and industry agreement on a minimum level of test effectiveness measure is needed. Error data at this level can then be compared from different programs. Analysis of such data will lead to a deeper understanding and analysis of software errors.

The Air Force is currently considering the level $k=2$, which corresponds to all branches being executed at least once, for adoption as an Air Force standard. (39, p.404).

Experience With Testing Large Programs. Significant experience with testing of large systems using this methodology is needed for uncovering empirical principles that can minimize the cost of the testing process and increase its effectiveness. Also empirical data can be used to define categories of specifications for a low, medium, or high level of testing.

AUTOMATED TOOLS

Most automated tools have been weak in two areas: user interface and volume of output. Interactive tools which provide the user with complete control of the volume of output, and which have a simplified user interface, are currently under development. The fourth-generation-software tools, as Ed Miller calls them, have been developed during the past few years. These tools have brought the theory of path testing into the domain of practical experience. Some of the more ambitious tools like SQLAB, JAVS, etc. provide a spectrum of support facilities (8,41,43,50,58,65). But they are not yet completely operational so that the Air Force can make their use mandatory in order to specify path testing for a program.

Ed Miller suggests the development of compilers which are better able to support the needs of generalized program analysis such as is needed in program testing activity. The PL/1 compiler goes a half step in this direction by producing a diagnostic table which contains, among other things, a complete symbol table. Some

of the needs of a program are simpler. For example, it would be desirable to have all the graph structures of a program produced automatically.

Advances in automated tools have focused on the test-data generation problem (4,25). Several automated aids have been proposed, but not much automatically generated data has been produced.

New automated tools which will significantly impact the application of path testing methodology to real programs should concentrate in the following areas:

Execution Verifiers and Coverage Analyzers. This tool category includes all forms of program instrumentation and associated dynamic behavior analysis to accomplish some form of automated coverage analysis of software systems under test. There have been many tools developed for this purpose, most of them for FORTRAN software. There is a need for well designed, efficient, and generalized portable coverage analyzers, as part of compilers or as pre/post-processing free-standing systems, which are applicable to a wide variety of languages and machines.

Test Harness Systems. A test harness system eases the pain of (i) setting up the test driver, (ii) generating appropriate stub information, (iii) verifying that the program produces correct results by comparison with predicted results, and (iv) keeping track of a series of test data. Existing packages claim to do this, but often place an unreasonably high burden of work on the program tester. An efficient and powerful test harness system is needed which eliminates the need for any work on the part of the program tester, except possibly for interactive-based queries by the system to the program tester.

Input/Output Generator. An input/output generator is a new tool that currently has no parallel in the tool community, but which does exist in prototype versions in several instances. The idea behind the tool is simple enough: given some input data for a program, determine which output variables are actually computed by, or are affected by, the program as it operates using the given input. The utility of this is that it is somewhat easier to use than a test harness, in which all of this information must be known by auxiliary means. In use, such a system would make it a simple matter to know, in general, how inputs relate to outputs. For complicated, multiple module, software systems, I/O generators should be developed to identify the outputs in symbolic form that are affected when particular inputs are supplied.

REFERENCES

1. D. S. Alberts, "The Economics of Software Quality Assurance," AFIPS Conf. Pro. 1976 NCC, Montvale, New Jersey, 1976. pp. 433-442.
2. T. Anderson, M. L. Shooman, and B. Randell, "Computing System Reliability" Cambridge Univ. Press, 1979.
3. R. E. Barlow, and F. Proschan, "Mathematical Theory of Reliability," New York. Wiley, 1965.
4. R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution," Proc., 1975 International Conf. on Reliable Software, Los Angeles, California, pp. 234-245.
5. F. P. Brooks, Jr., The Mythical Man-Month, Addison-Wesley, Reading, MA. 1975.
6. N. B. Brooks and E. N. Kostruba, "Advanced Software Quality Assurance: IFTRAN Preprocessor User's Manual," GRC, February 1978; ADB018436.
7. J. Brown, "Why Tools?," Proc. Eighth Annual Symp. on Computer Science and Statistics, UCLA, Los Angeles, February 1975.
8. J. R. Brown and M. Lipow, "Testing for Software Reliability," Proc., 1975 International Conf. on Reliable Software, Los Angeles, California, pp. 518-527.
9. T. A. Budd, R. DeMillo, and R. J. Lipton, "The Design of a Prototype Mutation System for Program Testing," 1978 National Computer Conference, June 1978.
10. J. N. Buxton and B. Randell, "Software Engineering Techniques," Brussels, Belgium: Scientific Affairs Div., NATO, Apr 1970.
11. J. D. Cooper and M. J. Fisher (editors), "Software Quality Management", New York: McGraw-Hill, 1979.

12. COMPUTER, April 1978.
13. D. R. Cox, "Renewal Theory" London: Methuen, 1962.
14. E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs. N. J. 1976.
15. E. W. Dijkstra, "Notes on structured programming," on O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Structured Programming, Academic Press, London, 1972.
16. E. W. Dijkstra, "Structured Programming" Software Engineering Techniques, NATO. Science Committee, Brussels, 1969, ed. by J. N. Buxton and B. Randell.
17. E. W. Dijkstra, "A constructive approach to the problem of program correctness," BIT 8, pp. 174-186, 1968.
18. M. E. Fagan, "Design and code inspections to reduce errors in program development," IBM Systems Journal 15, 2, pp. 182-211, 1976.
19. H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil, "On two problems in the generation of program test paths." IEEE Trans. Soft Eng., Vol. SE-2, p. 227-233 (Sept. 1976)
20. T. Gilb, "Software Metrics," Cambridge, MA; Winthrop, 1977.
21. J. B. Goodenough, "A Survey of Program Testing Issues," In Research Directions in Software Technology, MIT Press, 1979.
22. J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," Proc., 1975 International Conf. on Reliable Software, Los Angeles, California, pp. 493-510.
23. M. S. Hecit and J. D. Ullman, "Analysis of a Simple Algorithm for Global Data Flow Problems," Proc. AMC SIGACT/SIGPLAN Symp., Boston, MA. Oct. 1973.
24. R. V. Hogg and E. A. Tanis, "Probability and Statistical Inference," p. 426
25. M. Holthouse and E. S. Cosloy, "A Practical System for Automatic Testcase Generation," Presented at 1976 NCC, New York, June 1976.
26. W. E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing" SOFTWARE - Practice and Experience, Vol. 8, 381-397 (1978).

27. W. E. Howden, "Methodology for the Generation of Program Test Data," IEEE Trans. on Computer, Vol. C-24, No. 5, May 1975, 554-559.
28. W. E. Howden, "Reliability of the Path Analysis Testing Strategy," IEEE Trans. on Software Engineering, September 1976, pp. 208-214.
29. W. E. Howden, "A Survey of Dynamic Analysis Methods," Tutorial: Software Analysis and Validation Techniques, IEEE Comp. Soc. 1978, ed. by E. Miller and W. E. Howden, p. 185-206.
30. J. C. Huang, "An Approach to Program Testing" ACM Computing Surveys, Sept. 1975.
31. Info. Tech. Ltd., "State of the Art Report, Software Testing Vol. 1, Analysis and Bibliography; Vol. 2, Invited Papers," 1979.
32. Jensen and Wirth, PASCAL USER Manual and Report," 2nd edition, Springer, Verlag.
33. J. C. King, "A New Approach to Program Testing" Proc., 1975 International Conf. on Reliable Software. Los Angeles, California, pp. 228-233.
34. B. Littlewood, "MTBF is Meaningless in Software Reliability" IEEE Trans. Reliability, Vol. R-24, Apr. 1975, p. 82.
35. B. Littlewood, "How to Measure Software Reliability and How Not To" IEEE Trans. Reliability, Vol. R-28, No. 2, June 1979.
36. C. McGowan and R. McHenry, "Software Management", Research Direction in Software Technology, Ed. by P. Wagner, The MIT Press, 1979m pp. 207-253.
37. R. C. McGowan, "A Proposed Software Development Process," Proc. of the 10th Annual International Hawaii Conference on Systems Sciences, January 1977.
38. H. Mills, "Top-Down Programming in Large Systems," in Debugging Techniques in Large Systems. R. Rustin (ed.), Prentice-Hall, Englewood Cliffs, NJ, pp. 41-45, 1971.
39. E. F. Miller, "Program Testing Technology in the 1980s" The Oregon Report: Proceedings of the IEEE Conf. on Computing in the 1980s, 1978.

40. E. F. Miller, "Program Testing: Art Meets Theory," IEEE Computer, July, 1977.
41. E. F. Miller, "A General Purpose Program Analyzer," Software Research Associates, RN-210, March 1977.
42. E. F. Miller, et al., "Jovial Automated Verification System," Rome Air Development Center, RADC-TR-76-20, Feb. 1976, AD A022 056.
43. E. F. Miller, "RXVP-An Automated Verification System for FORTRAN," Proc. Workshop 4, Computer Science and Statistics: Eighth Annual Symposium on the Interface, Los Angeles, California, February 1975.
44. E. F. Miller, M. R. Paige, J. P. Benson, and W. R. Wisehart, "Structural Techniques of Program Validation Proc." International Conf. on Reliable Software, IEEE 1975.
45. J. D. Musa, "Validity of Execution-Time Theory of Software Reliability" IEEE Trans. on Reliability, Vol. R-28, No. 3, Aug. 1979, p. 181-191.
46. G. J. Myers, "The Art of Software Testing", Wiley-Interscience Publication, 1979.
47. G. J. Myers, "Software Reliability" Wiley-Interscience Publication, 1976.
48. W. Myers, "COMSEC 78 Wrap-Up" Computer, IEEE, New York, Jan. 1979.
49. L. J. Osterweil, "Dave--A Validation Error Detection and Documentation System for FORTRAN," Software Practice and Experience, Vol. 6, 1976.
50. L. J. Osterweil and L. D. Fosdick, "Data Flow Analysis as an Aid to Documentation, Assertion, Assertion Generation, Validation, and Error Detection," CU-CS-055-74, University of Colorado, September 1974.
51. M. R. Paige, "A Pragmatic Approach to Software Testcase Generation," Science Applications, Inc., September 1975.
52. M. R. Paige, "Software Reliability Techniques," Reliability Chapter, Boston Section, IEEE State-of-the-Art Seminar, presented Oct. 1979.

53. M. R. Paige, "Program Graphs, an Algebra and Their Implication for Programming", IEEE Transactions on Software Engineering, September 1975, 286-291.
54. W. Ross, "Structured Programming; Guest Editor's Introduction," Computer, Vol. 8, No. 6, pp. 21-22, 1975.
55. N. F. Schneidwind, "Application of Program Graphs and Complexity Analysis to Software Development and Testing," IEEE Transactions on Reliability, Vol. R-28, No. 3, Aug. 1979, P. 192-98.
56. H. A. Simon, "The Sciences of the Artificial" Cambridge, MA., MIT Press. 1969.
57. L. G. Stucki, "New Directions in Automated Tools for Improving Software Quality" Current Trends in Programming Methodology, Vol. 2 by Raymond T. Yeh, Prentice Hall, Inc. 1977.
58. L. G. Stucki, " A Prototype Automatic Program Testing Tool," AFIPS Conf. Proc., 1972 FJCC, Montvale, New Jersey, 1972, pp. 829-836.
59. J. E. Sullivan, "Measuring the Complexity of Computer Software," The MITRE Corporation Report, 1973, AD A007 770.
60. R. H. Thayer, "Rome Air Development Center R&E Computer Program in computer Language Controls and Software Engineering Techniques RADC-TR-74-80, 1974, AD 778 836.
61. E. Ulsamer, "Computers - Key to Tomorrow's Air Force," AIR FORCE Magazine, 56 (7), 46-52 (1973).
62. Martin Woodward, David Hedley, and Michael Hemmell, "Experience Software with Path Analysis and Testing of Programs"; IEEE Trans. of Softwear Eng., Vol. SE-6, No. 3, May 1980 pp. 277-286.
63. R. T. Yeh (Editor), "Current Trends in Programming Methodology Vol. II: Program Validation," Prentice-Hall, 1977.
64. F. Zurcher and B. Randell, "Iterative multi-level modeling - a methodology for computer system design design," Proc. IFIP Congress 1968, Booklet D. North Holland Publ, Co. Amsterdam, pp. 138-142, 1968.
65. M. R. Paige, "A Methodology for Generating Paths in a Directed Graph" GRC IM-1816m Biv, 1973.

GLOSSARY OF ACRONYMS

AF	Air Force
ASPR	Armed Services Procurement Regulation
BMD	Ballistic Missile Centre
BISS	Base Installation Security System
CDR	Critical Design Review
CPC	Computer Program Component
CPCI	Computer Program Configuration Item
CPU	Central Processing Unit
DAR	Defense Acquisition Regulation
DD Path	Decision-to-Decision Path
DEM	Data Effectiveness Measure
DoD	Department of Defense
DoDD	Department of Defense Directive
ECS	Entry Control System
FY	Fiscal Year
GRC	General Research Corporation
HIPO	Hierarchical Input-Process-Output
IBM	International Business Machines
I/O	Input/Output
JAVS	JOVIAL Automatic Verification System
MTBF	Mean-Time-Between Failure
MTTR	Mean-Time-To-Repair

NASA	National Aeronautics and Space Administration
PDR	Preliminary Design Review
R&D	Research and Development
RADC	Rome Air Development Centre
RAM	Reliability, Availability, Maintainability
QA	Quality Assurance
SQA	Software Quality Assurance
SQLAB	Software Quality Assurance Laboratory
TEM	Test Effectiveness Measure
VCG	Verifiable Condition Generator
VPASCAL	Verifiable PASCAL

4
END
86